Predictable Data Communications with (Self-)Adjusting Networks

Stefan Schmid (University of Vienna, Austria)

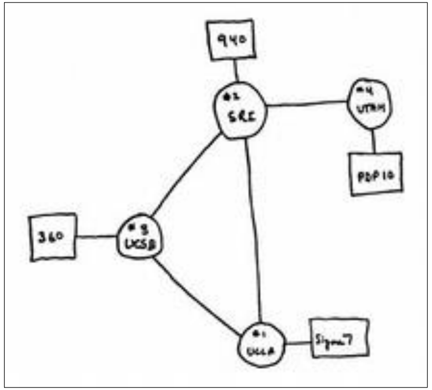
Predictable Data Communications with (Self-)Adjusting Networks

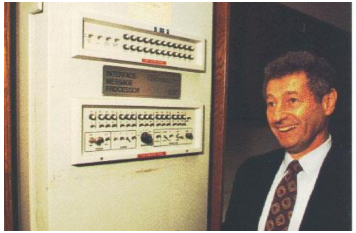
Stefan Schmid *et al.*, ideas from, e.g.,: Chen Avin (BGU, Israel) and Jiri Srba (Aalborg University, Denmark)

Predictable Data Communications with (Self-)Adjusting Networks

Stefan Schmid *et al.*, more recently also: Bruna Peres, Olga Goussevskaia, Kaushik Mondal

Networks and requirements have evolved…

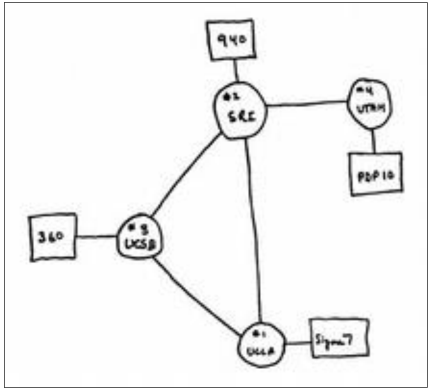


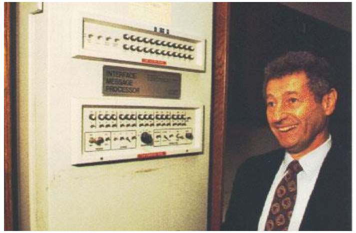


Early Internet users:

Kleinrock

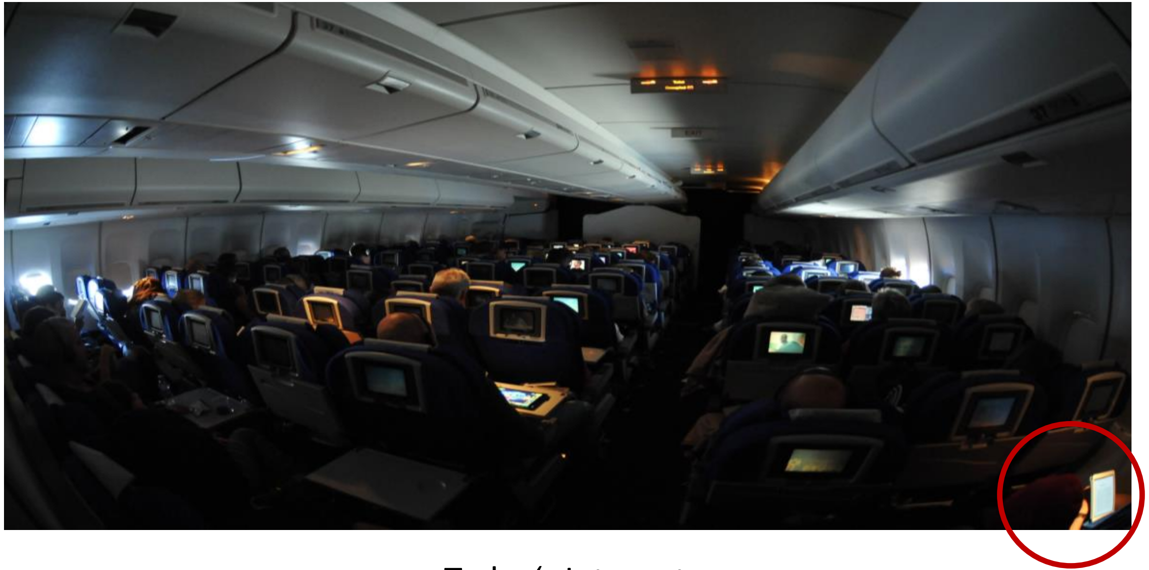
Networks and requirements have evolved…





Early Internet users:

Kleinrock



Today‘s Internet users

Credits: Oliver Hohlfeld

QoE: The Network Matters

- Trend toward **data-centric** …
  - Social networks, multimedia, financial services, …

- … **network-hungry** applications
  - Batch processing, streaming, scale-out DBs, ***distributed machine learning***, …

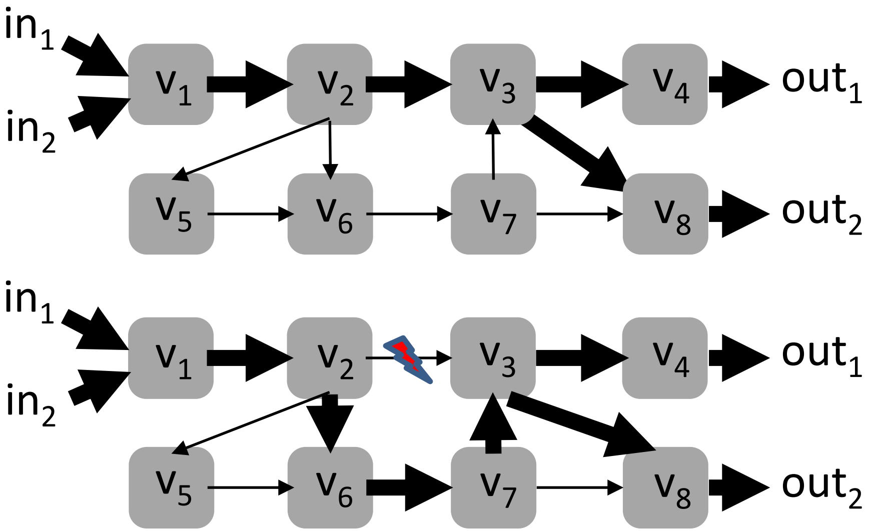
- Application performance and QoE critically **depend on network**

How to Provide Predictable Performance If

How to Provide Predictable Performance If

- ***Application performance*** critically depends on network…

- … but there can be **failures**?



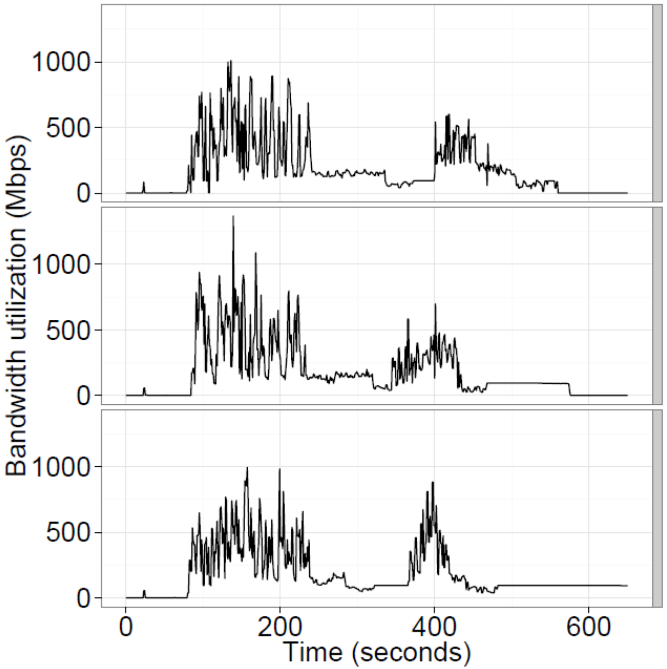
Complex failover (especially if distributed): packet reordering, timeouts, disconnect?

How to Provide Predictable Performance If

- ***Application performance*** critically depends on network…

- … but there can be **failures**?

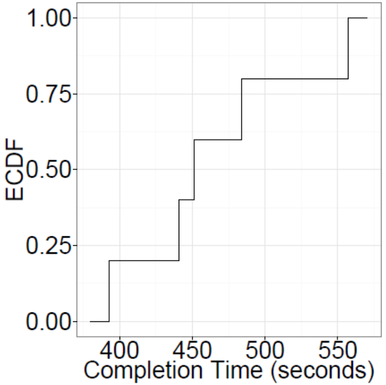
- … bandwidth **demand** is unpredictable?



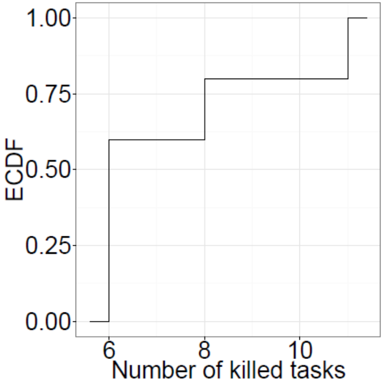
Complex congestion control? Idealized!

How to Provide Predictable Performance If

- ***Application performance*** critically depends on network…
- … but there can be **failures**?
- … bandwidth **demand** is unpredictable?
- … **executions** are unpredictable?



>20% variance in runtime

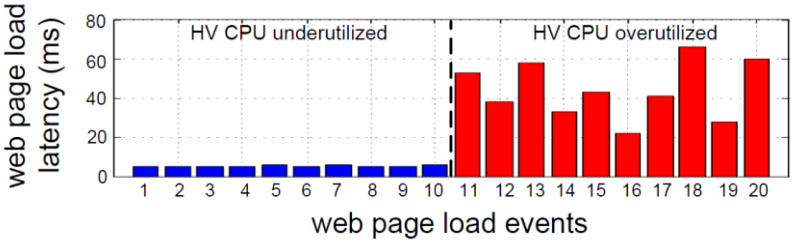


>50% variance in speculated tasks

Complex algorithms! E.g., speculation.

How to Provide Predictable Performance If

- ***Application performance*** critically depends on network…
- … but there can be **failures**?
- … bandwidth **demand** is unpredictable?
- … **executions** are unpredictable?
- … systems / **models** are complex?



E.g., web page load latency depends on network hypervisor!

Roadmap

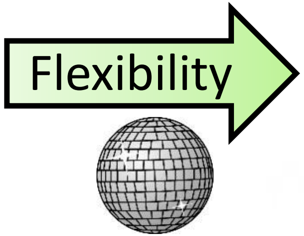
Roadmap

- Predictable performance under uncertainty is hard

Nils Bohr

"Prediction is difficult, especially about the future."

- Observation: at the same time, networks become more **flexible**! Idea: exploit for ***predictability***…



- … but it can be ***hard for humans***:

a case for **formal methods**? *Hot* right now (and here!)

- … but that can even be ***hard for computers***: so?!

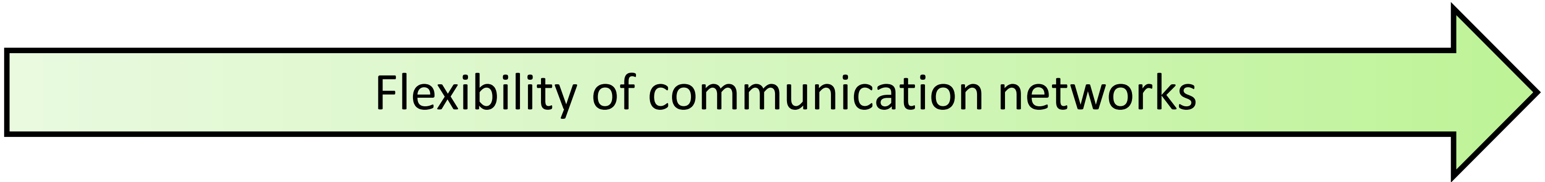
Especially **quantitative aspects** but important for QoE!

Ensuring Predictable Performance Under Uncertainty is Hard

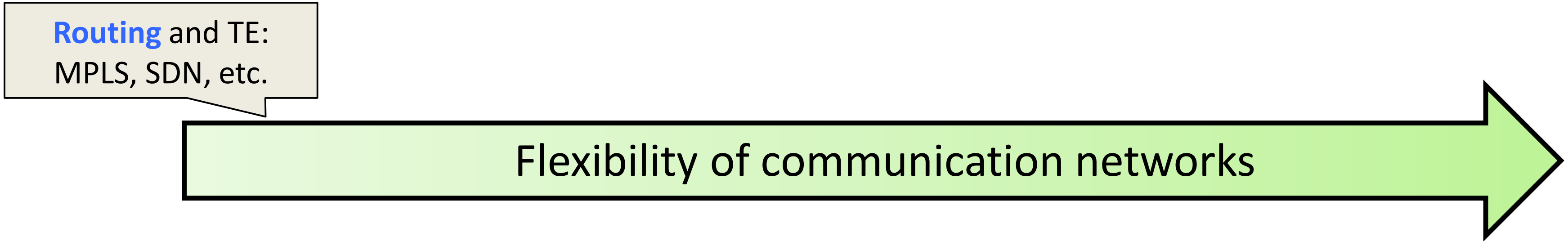
Ensuring Predictable Performance Under Uncertainty is Hard

Proposal: Exploit flexibilities!

Self-adjust to compensate and improve.



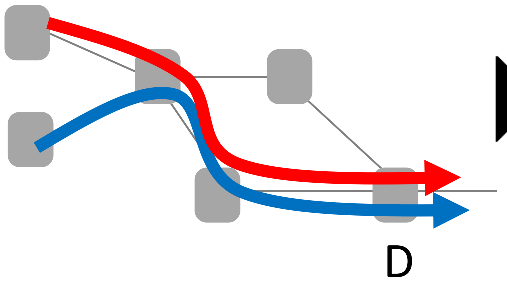
Flexibility of communication networks



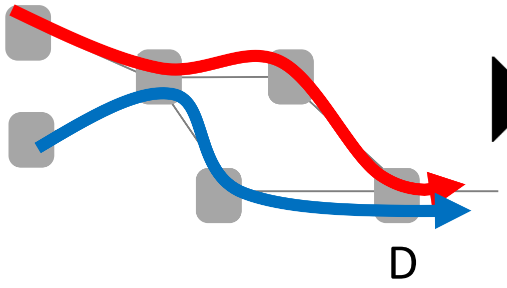
Routing and TE:
MPLS, SDN, etc.

Flexibility of communication networks

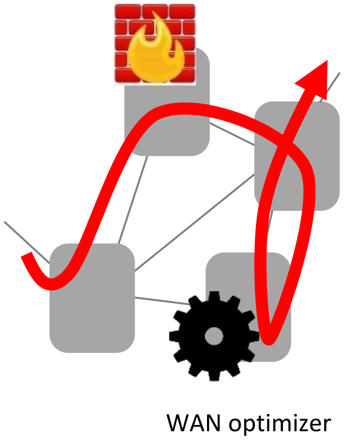
- Destination-based
- Shortest paths



- Arbitrary paths
- “simple paths”



- Waypoint routing
- Application-aware (TCP port)



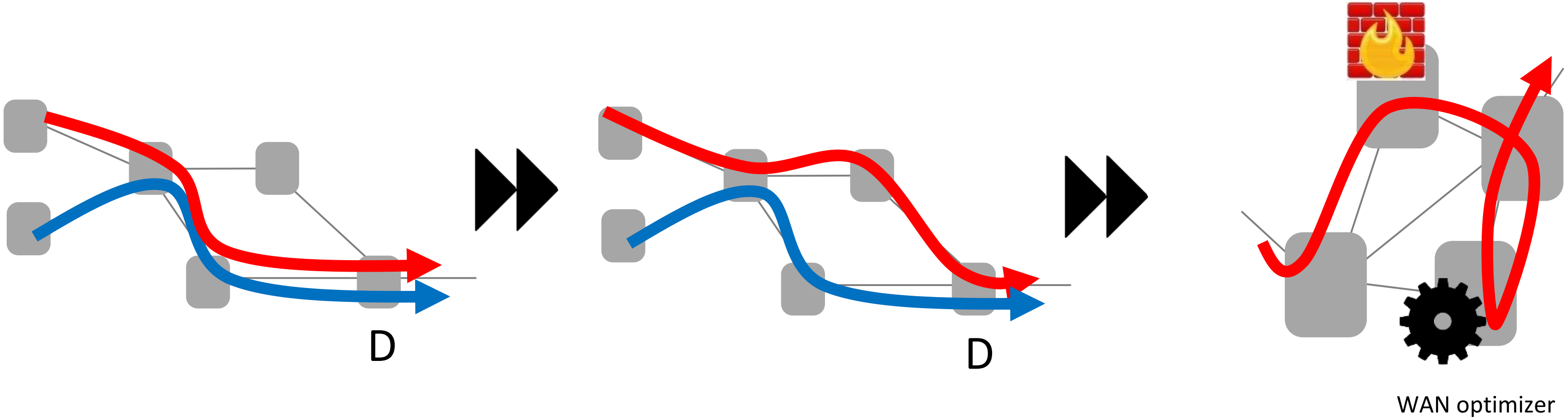
Routing and TE:
MPLS, SDN, etc.

Flexibility of communication networks

- Destination-based
- Shortest paths

- Arbitrary paths
- “simple paths”

- Waypoint routing
- Application-aware (TCP port)



Routing and TE:
MPLS, SDN, etc.

More alternatives routes, more capacity, etc.

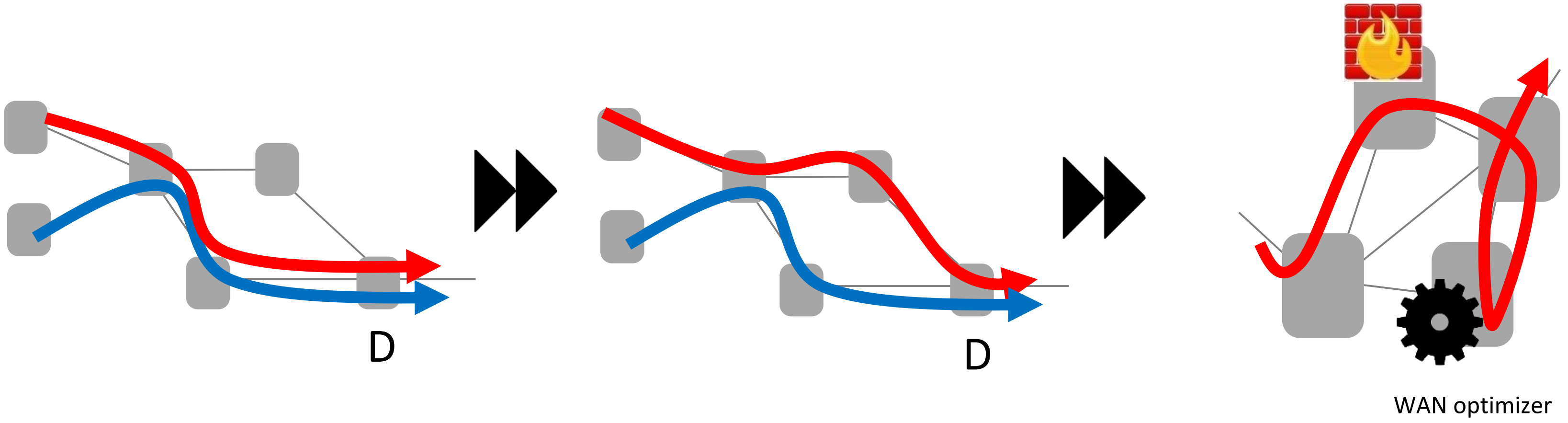
Flexibility of communication networks

Tomographic Node Placement Strategies and the Impact of the Routing Model. **SIGMETRICS** 2018.

Charting the Algorithmic Complexity of Waypoint Routing. **SIGCOMM CCR** 2018.

- Destination-based
- Shortest paths

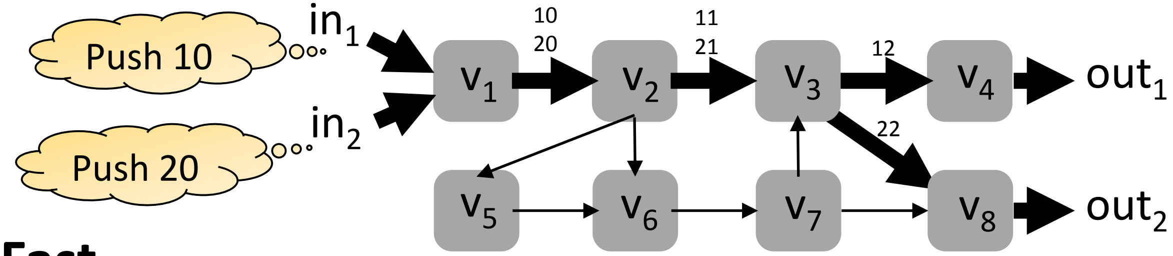
- Arbitrary paths
- “simple paths”



Routing and TE: MPLS, SDN, etc.

More alternatives routes, more capacity, etc.

Flexibility of communication networks

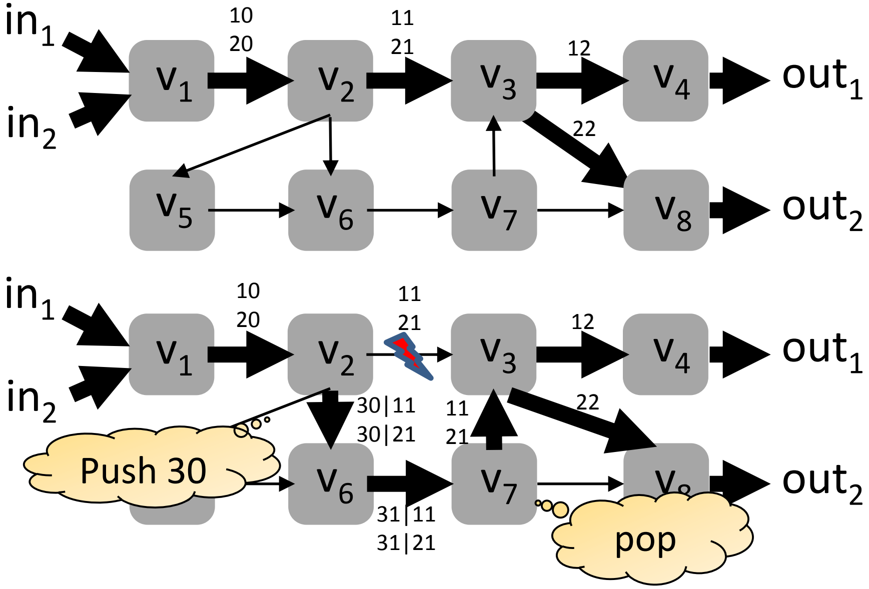


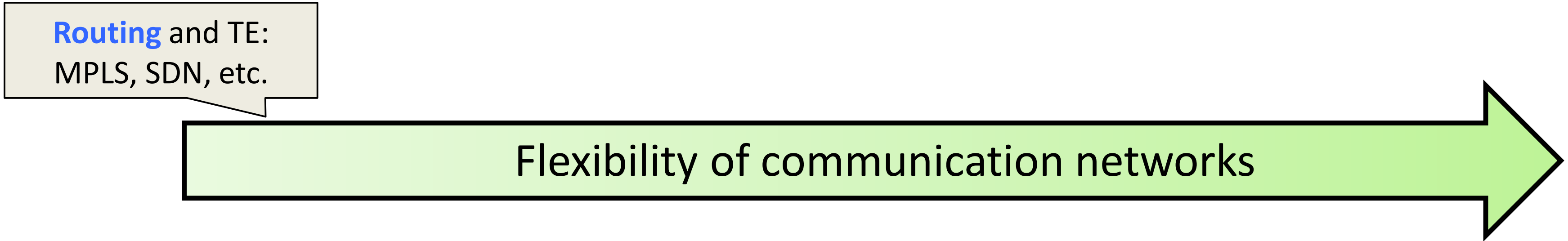
Also: flexible Fast Re-Routing (FRR) algorithms

Routing and TE:
MPLS, SDN, etc.

Flexibility of communication networks

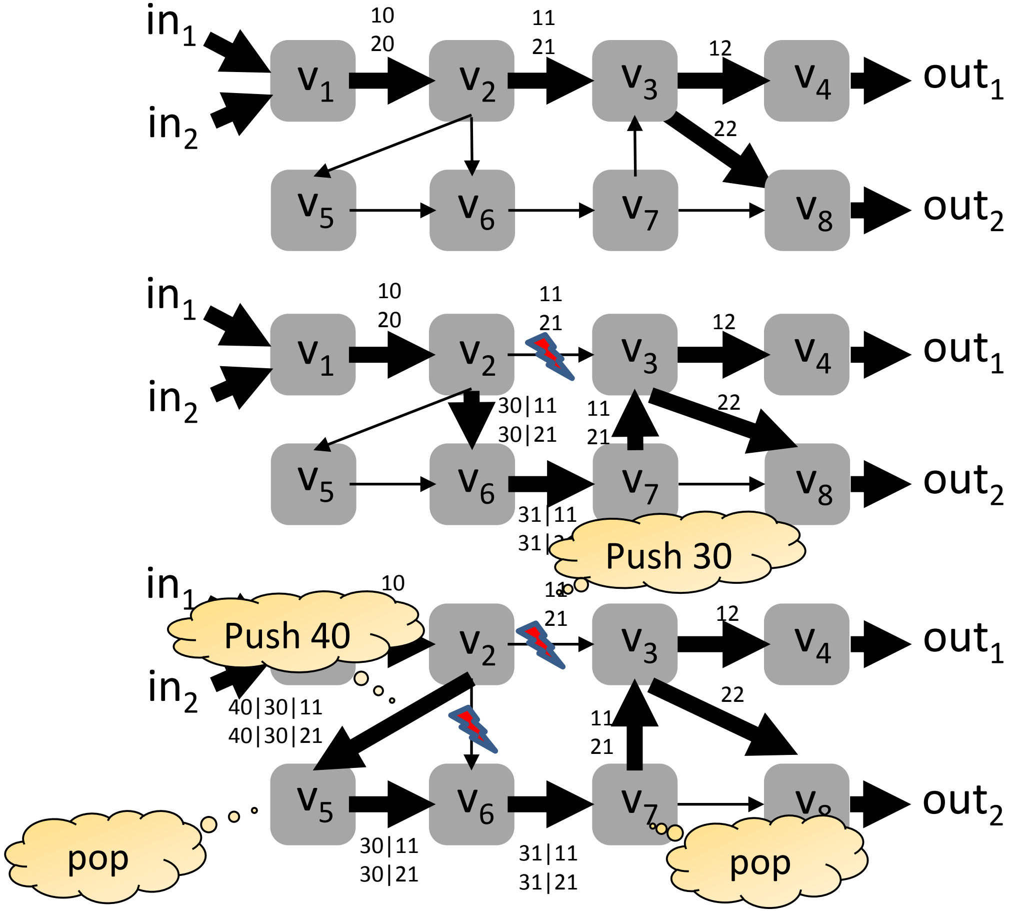
Also: flexible Fast Re-Routing (FRR) algorithms





Routing and TE: MPLS, SDN, etc.
Flexibility of communication networks

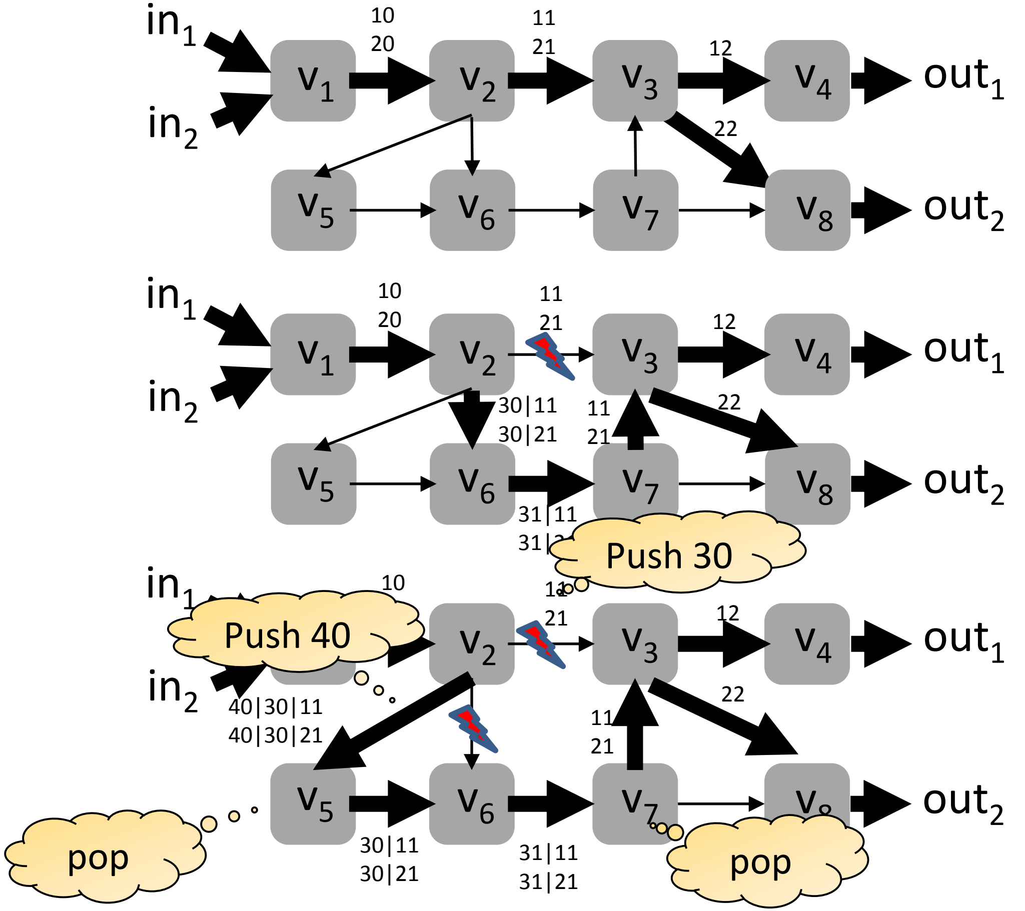
Also: flexible Fast Re-Routing (FRR) algorithms



Routing and TE: MPLS, SDN, etc.

Flexibility of communication networks

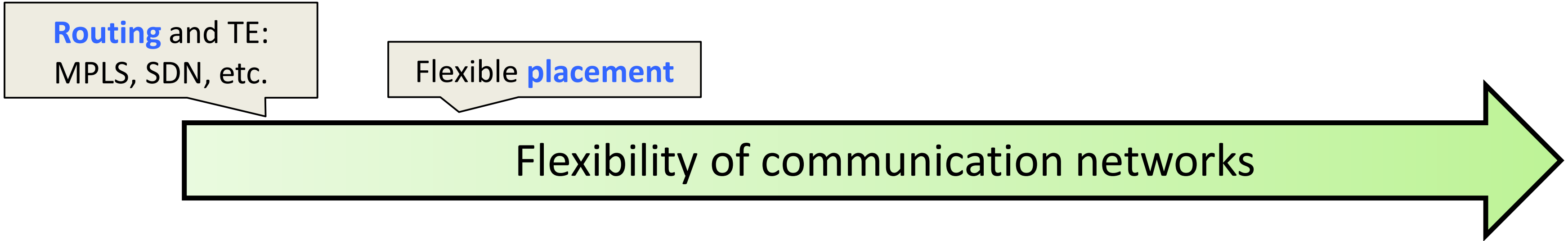
Also: flexible Fast Re-Routing (FRR) algorithms



Fast & high capacity!

Routing and TE:
MPLS, SDN, etc.

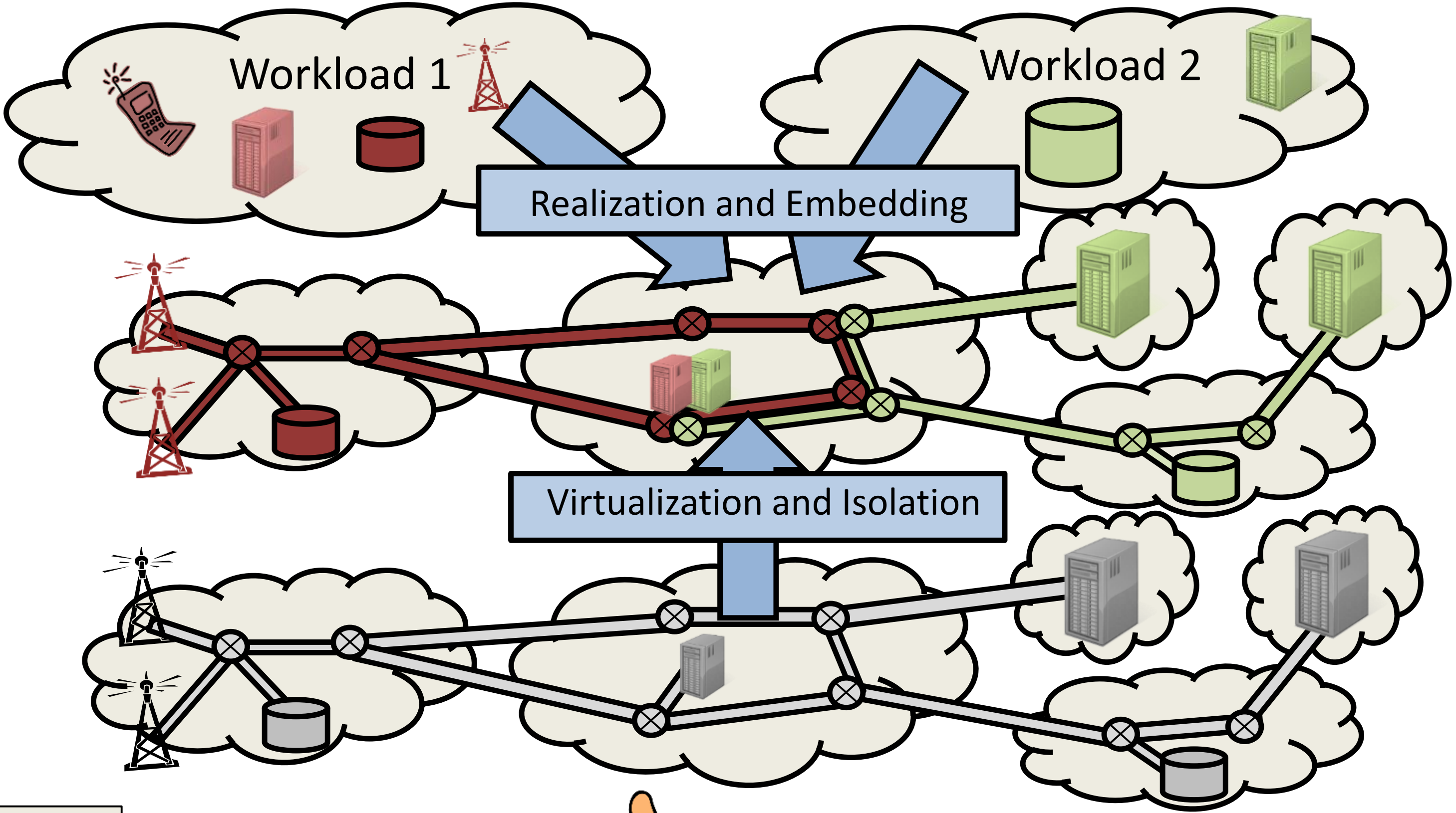
Flexibility of communication networks



Routing and TE:
MPLS, SDN, etc.

Flexible **placement**

Flexibility of communication networks



Workload 1

Workload 2

Realization and Embedding

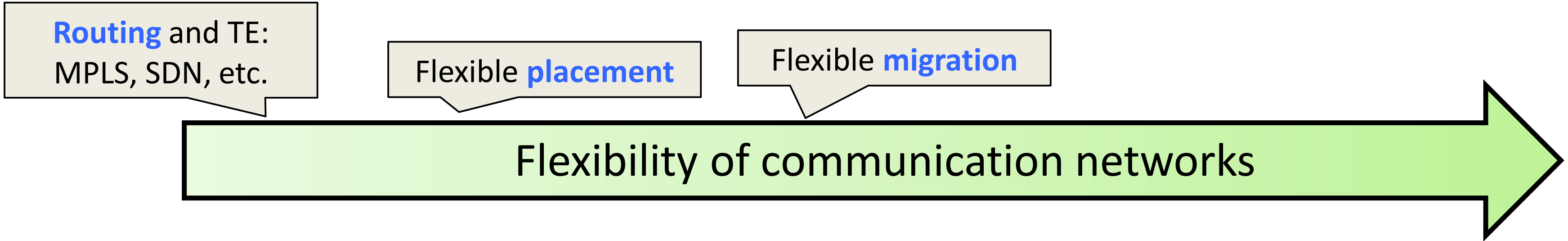
Virtualization and Isolation

Routing and TE: MPLS, SDN, etc.

Flexible placement

Improved utilization

Flexibility of communication networks



Routing and TE: MPLS, SDN, etc.

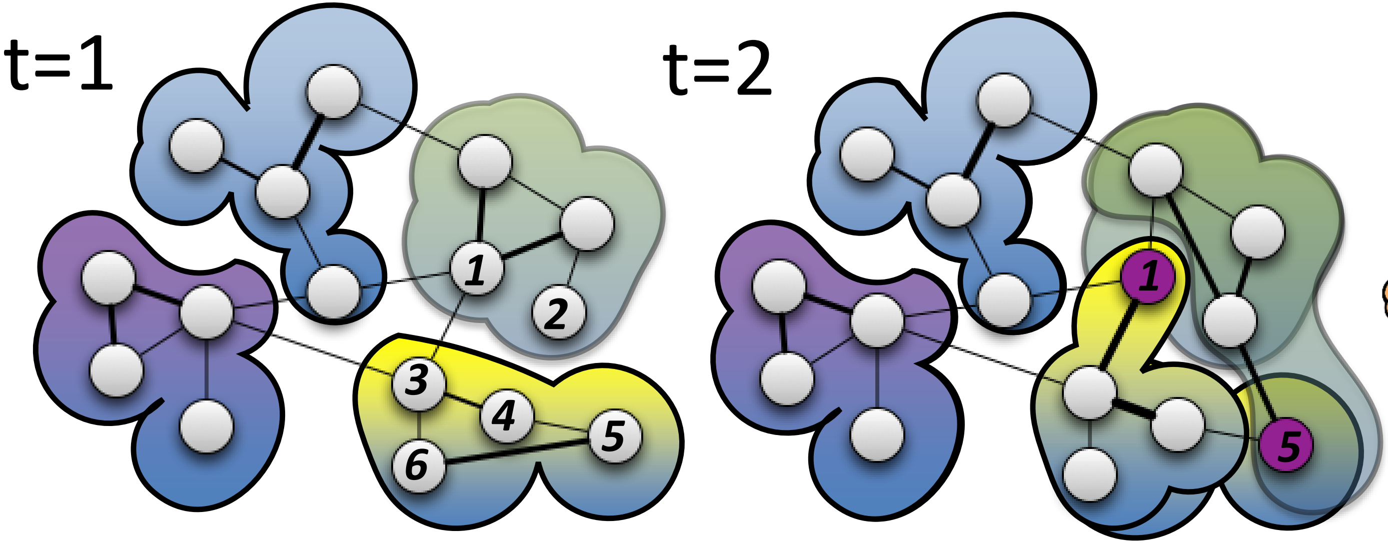
Flexible **placement**

Flexible **migration**

Flexibility of communication networks

Communication Graph (e.g., VMs on servers with 4 cores):

If more communication (1,3),(3,4),(2,5) but less (5,6): migrate!



Reduced resource consumption, better latency, etc.

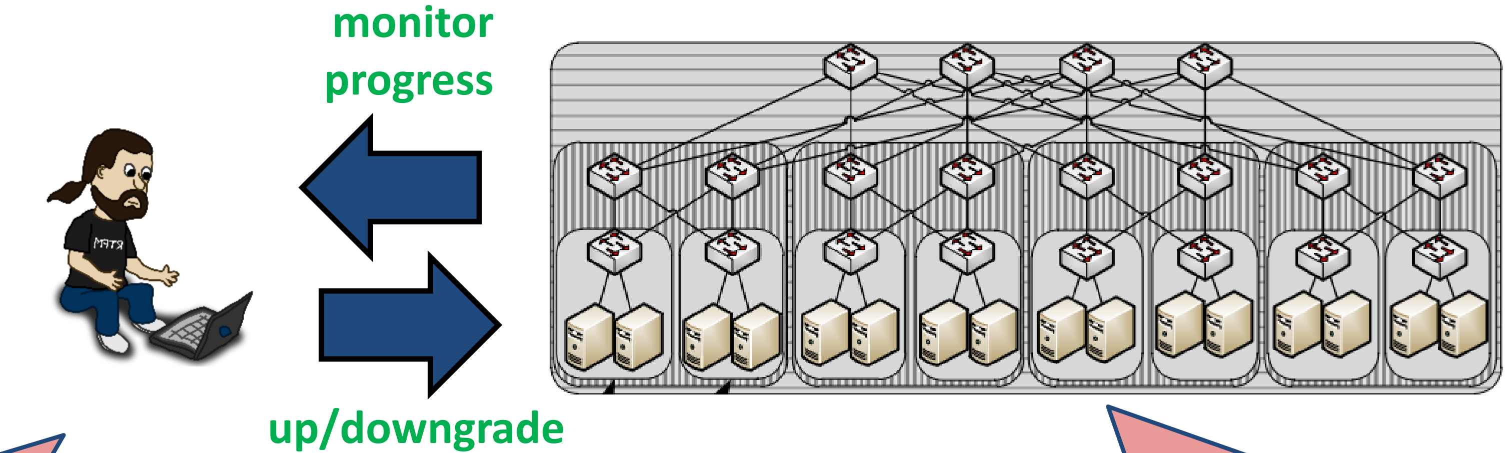
Routing and TE: MPLS, SDN, etc.

Flexible placement

Flexible migration

Flexibility of communication networks

Kraken: Dynamic scale-out / scale-in (requires migration)



Monitor and react according to performance needs.

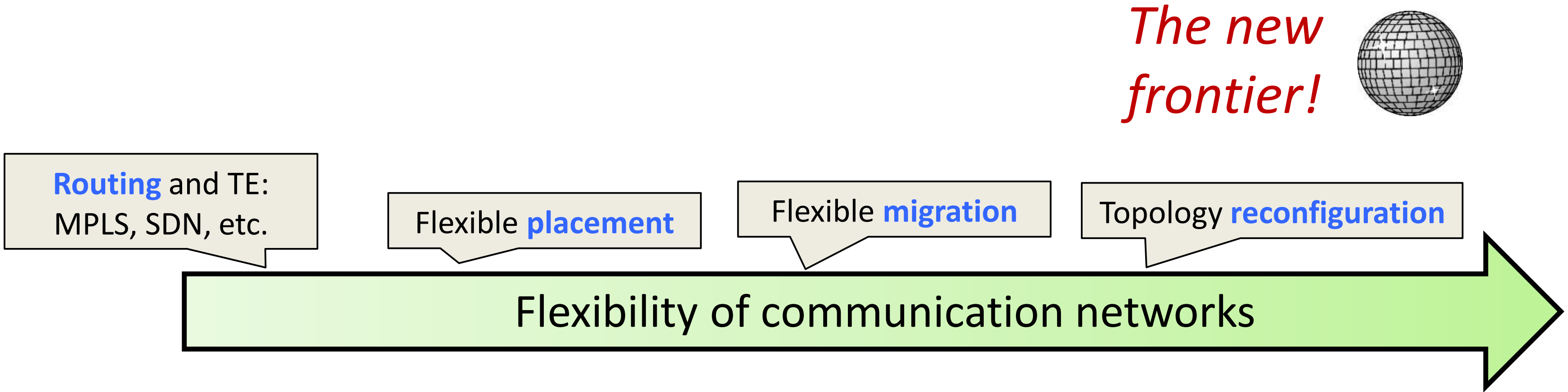
Automatic reconfiguration given current resources demand.

Routing and TE: MPLS, SDN, etc.

Flexible placement

Flexible migration

Flexibility of communication networks



The new frontier!

Routing and TE: MPLS, SDN, etc.

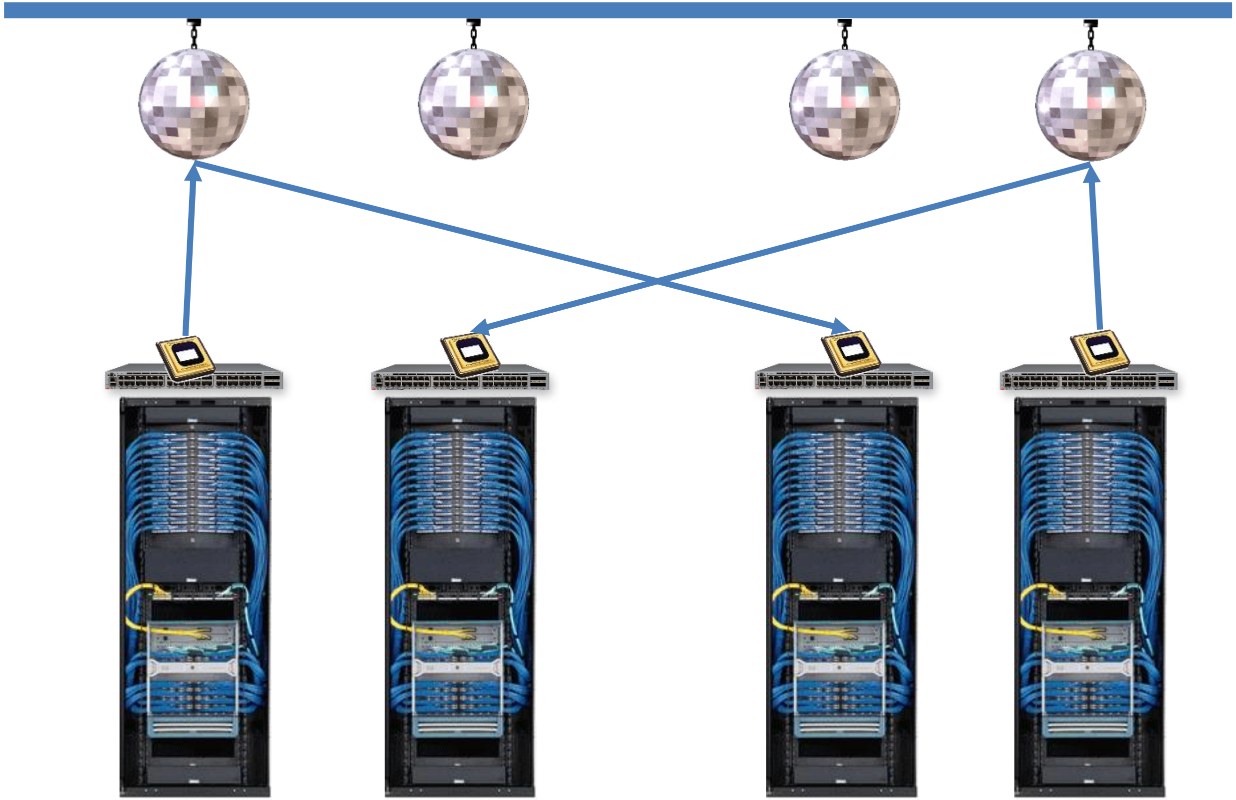
Flexible **placement**

Flexible **migration**

Topology **reconfiguration**

Flexibility of communication networks

t=1



Routing and TE: MPLS, SDN, etc.

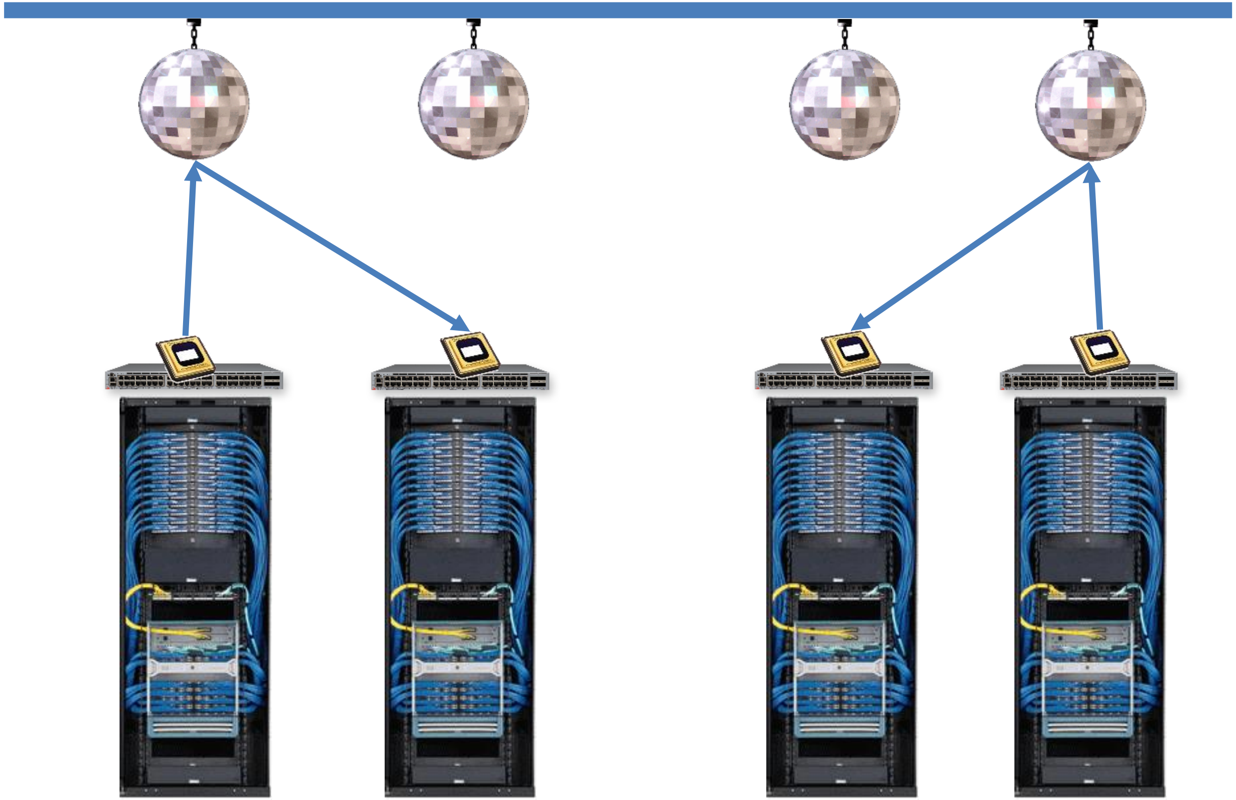
Flexible placement

Flexible migration

Topology reconfiguration

Flexibility of communication networks

t=2



Routing and TE: MPLS, SDN, etc.

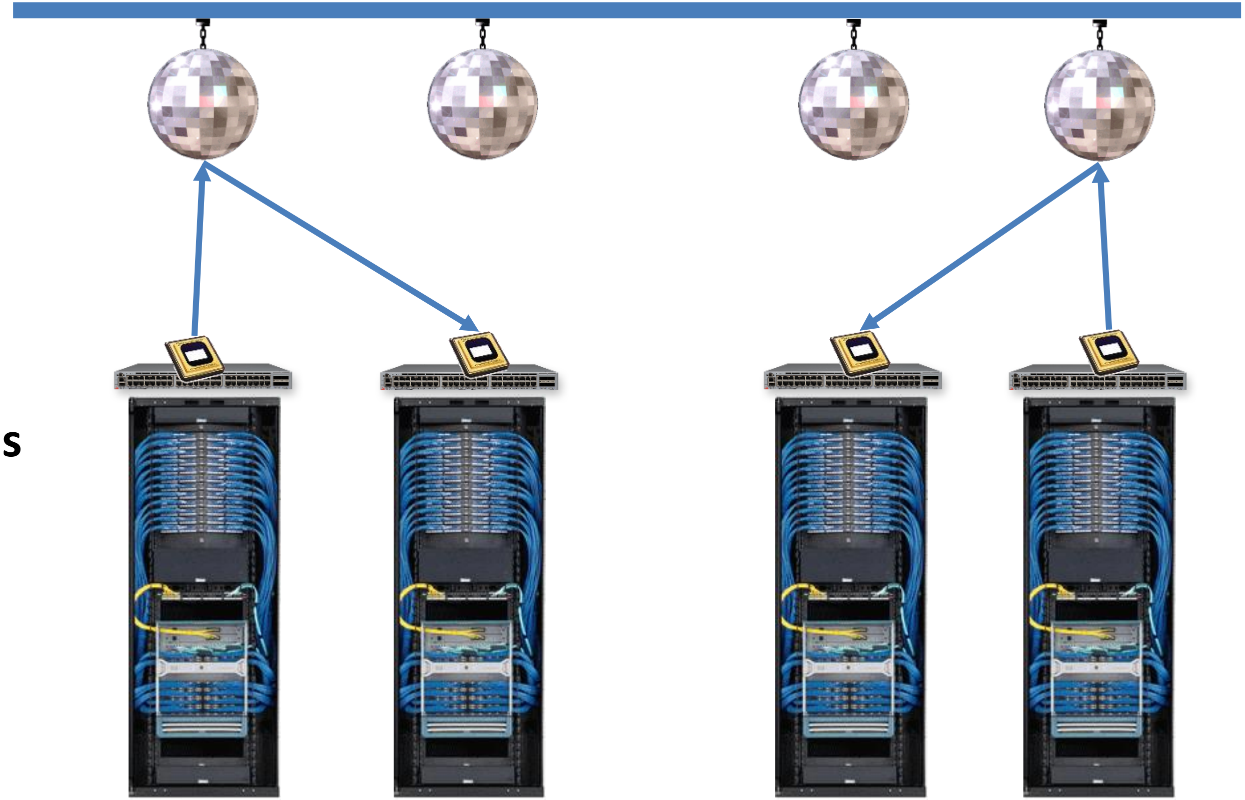
Flexible placement

Flexible migration

Topology reconfiguration

Flexibility of communication networks

t=2



Less resources and latency

Routing and TE:
MPLS, SDN, etc.

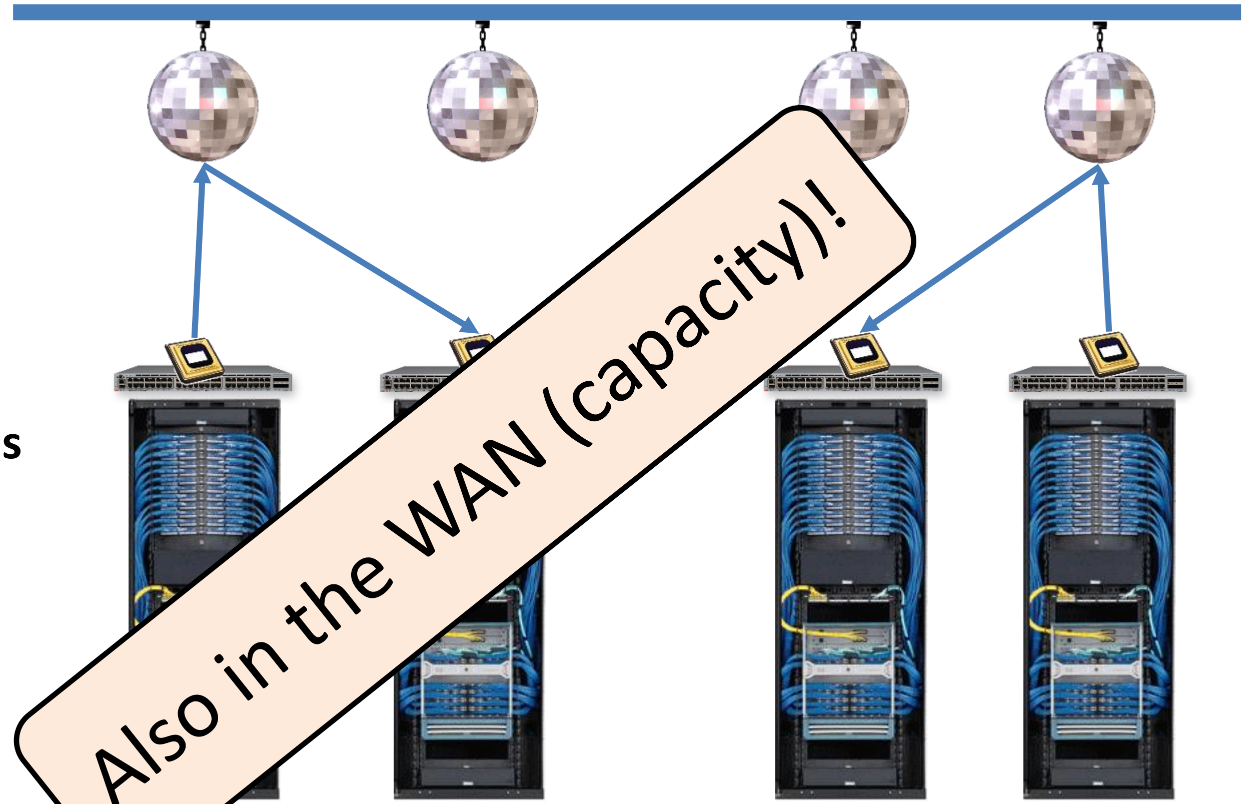
Flexible placement

Flexible migration

Topology reconfiguration

Flexibility of communication networks

t=2



Less resources and latency

Also in the WAN (capacity)!

Routing and TE: MPLS, SDN, etc.

Flexible **placement**

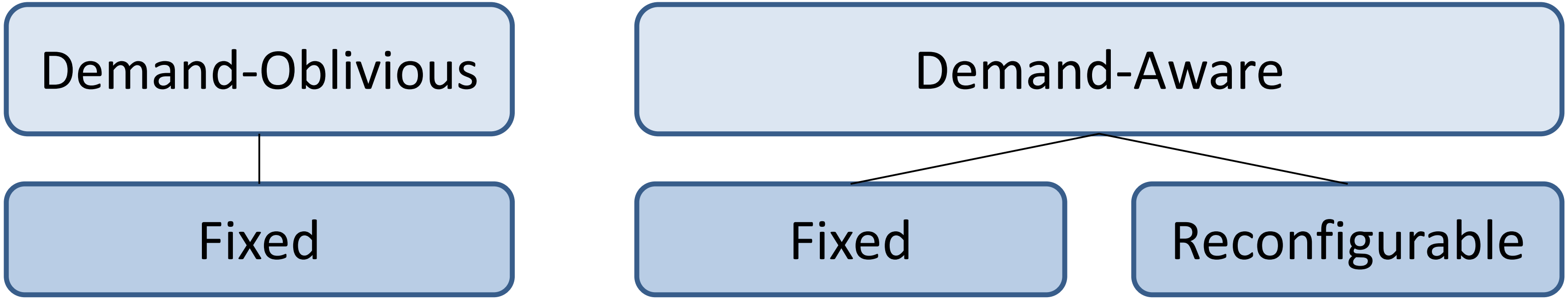
Flexible **migration**

Topology **reconfiguration**

Flexibility of communication networks

Since this is the latest trend, let's have a closer look:

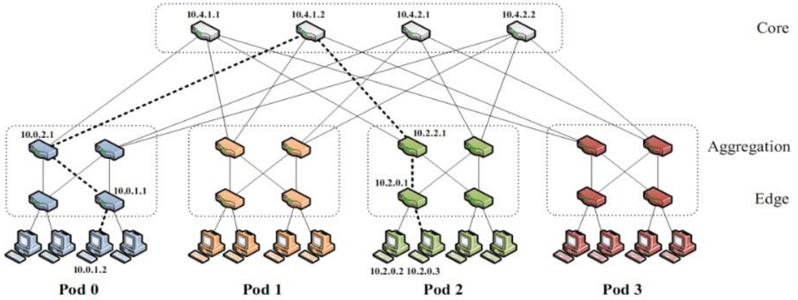
A Brief History of Self-Adjusting Networks

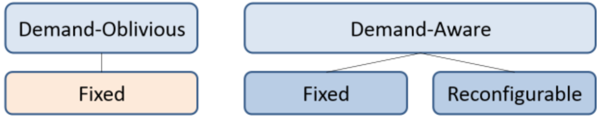


*Focus on **datacenters** but more general…*

Traditional Networks

- Lower bounds and hard **trade-offs**, e.g., degree vs diameter
- Usually optimized for the "worst-case" (**all-to-all** communication)
- Example, fat-tree topologies: provide **full bisection bandwidth**




Demand-Oblivious

Fixed

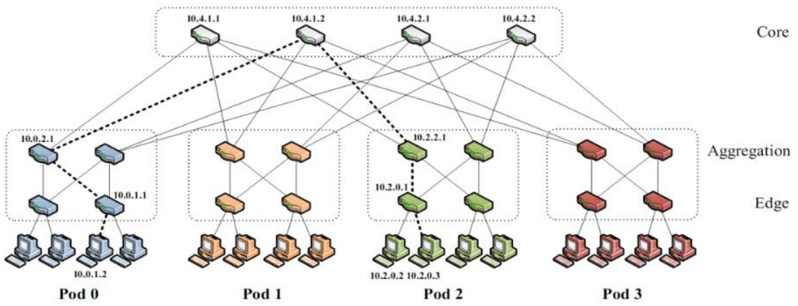
Demand-Aware

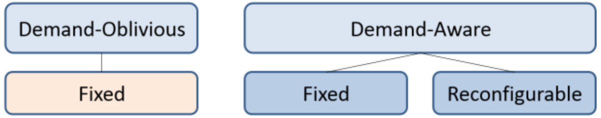
Fixed

Reconfigurable

Traditional Networks

- Lower bounds and hard **trade-offs**, e.g., degree vs diameter
- Usually optimized for the “worst-case” (**all-to-all** communication)
- Example, fat-tree topologies: provide **full bisection bandwidth**





Demand-Oblivious

Fixed

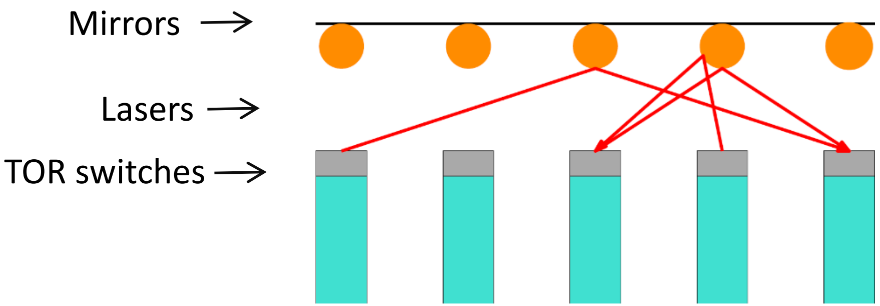
Demand-Aware

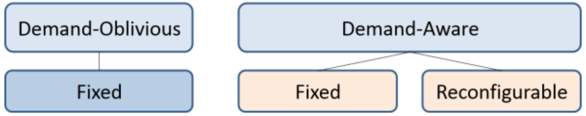
Fixed

Reconfigurable

Vision: DANs and SANs

- ***DAN***: Demand-Aware Network
  - Statically optimized **toward the demand**
- ***SAN***: Self-Adjusting Network
  - **Dynamically optimized toward** the (time-varying) demand





Demand-Oblivious

Fixed

Demand-Aware

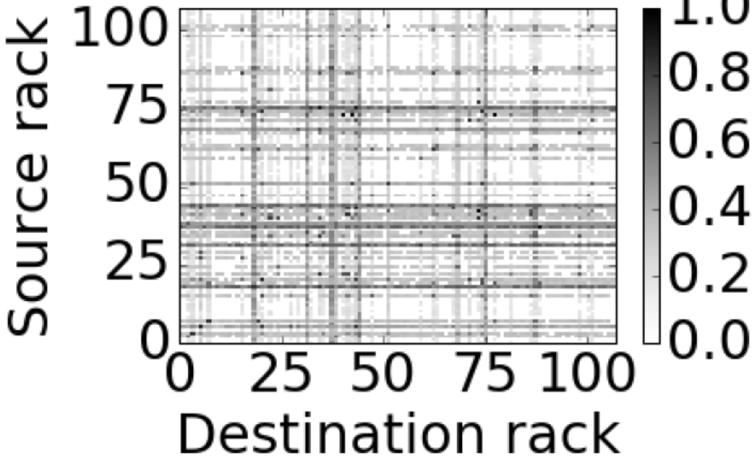
Fixed

Reconfigurable

Empirical Motivation

- Real traffic pattners are far from random: ***sparse*** structure

Structure :-)



Heatmap of rack-to-rack traffic
ProjecToR @ SIGCOMM 2016

Empirical Motivation

- Real traffic pattners are far from random: ***sparse*** structure

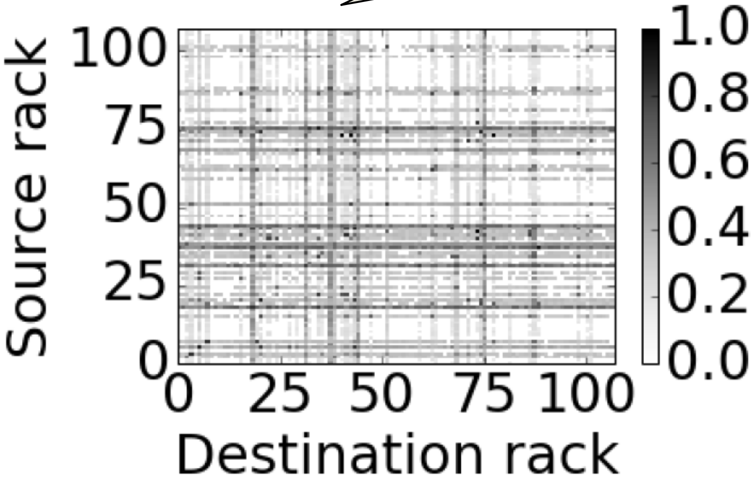
- Little to no communication between certain nodes

  A case for **DAN**s!

- But also ***changes*** over time

  A case for **SAN**s!

Structure :-)



Heatmap of rack-to-rack traffic
ProjecToR @ SIGCOMM 2016

Analogous to *Datastructures*: Oblivious…

- Traditional, **fixed** BSTs do not rely on any assumptions on the demand

- Optimize for the **worst-case**

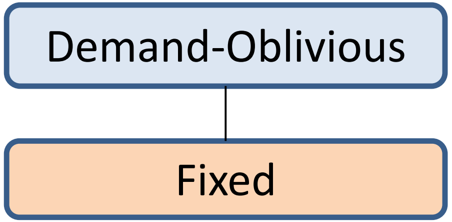
- Example **demand**:

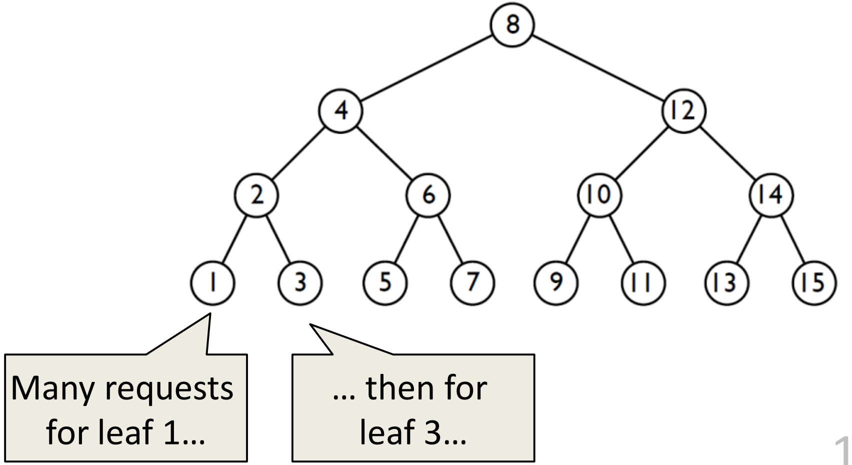
  1,…,1,3,…,3,5,…,5,7,…,7,…,log(n),…,log(n)

  ⟷ ⟷ ⟷ ⟷ ⟷

  many many many many many

- Items stored at ***O(log n)*** from the root, **uniformly** and **independently** of their frequency





Many requests for leaf 1…

… then for leaf 3…

Analogous to *Datastructures*: Oblivious…

- Traditional, **fixed** BSTs do not rely on any assumptions on the demand

- Optimize for the **worst-case**

- Example **demand**:

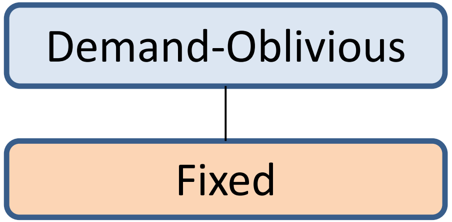
  1,…,1,3,…,3,5,…,5,7,…,7

  ⟷ ⟷ ⟷ ⟷

  many *many* *many* *many*

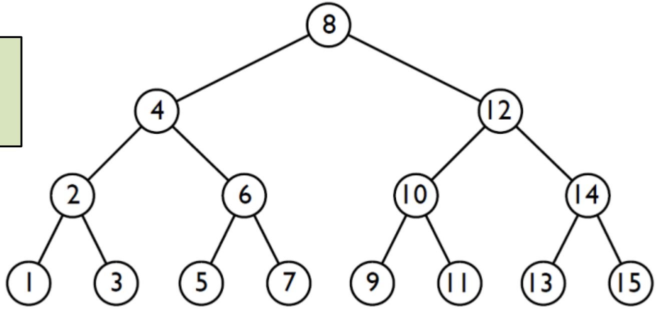
Amortized cost corresponds to ***max entropy of demand***!

- Items stored at ***O(log n)*** from the root, **uniformly** and **independently** of their frequency



Demand-Oblivious

Fixed



Many requests for leaf 1…

… then for leaf 3…

… Demand-Aware …

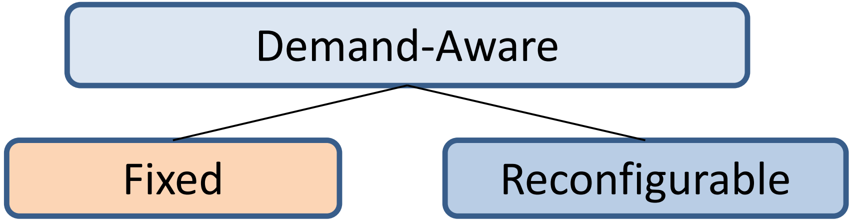
- **Demand-aware fixed** BSTs can take advantage of ***spatial locality*** of the demand

- Optimize: place frequently accessed elements close to the root
  - Recall example **demand**: 1,…,1,3,…,3,5,…,5,7,…,7,…,log(n),…,log(n)

- E.g., **Mehlhorn** trees

Amortized cost corresponds to ***empirical entropy of demand***!

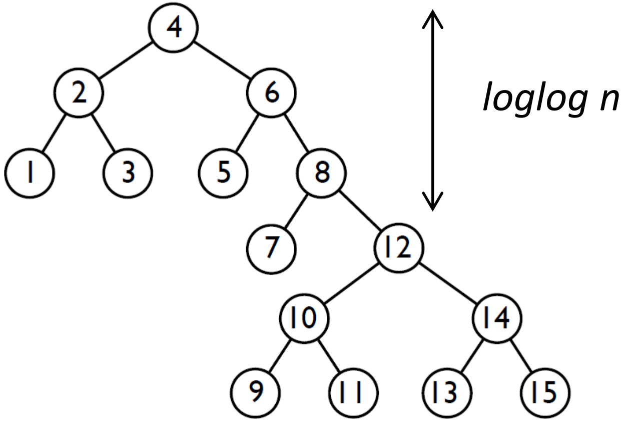
- Amortized cost ***O(loglog n)***



Demand-Aware

Fixed

Reconfigurable

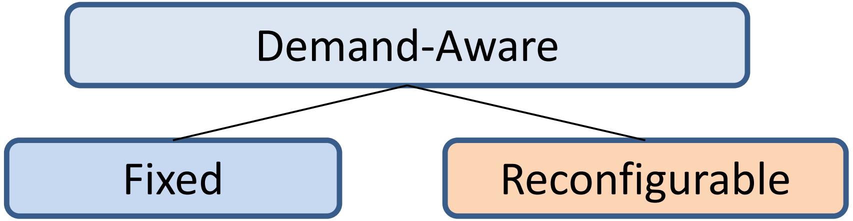


… Self-Adjusting!

- **Demand-aware reconfigurable** BSTs can additionally take advantage of ***temporal locality***

- By moving accessed element to the root: amortized cost is ***constant***, i.e., O(1)
  - Recall example **demand**: 1,…,1,3,…,3,5,…,5,7,…,7,…,log(n),…,log(n)

- Self-adjusting BSTs e.g., useful for implementing ***caches*** or garbage collection

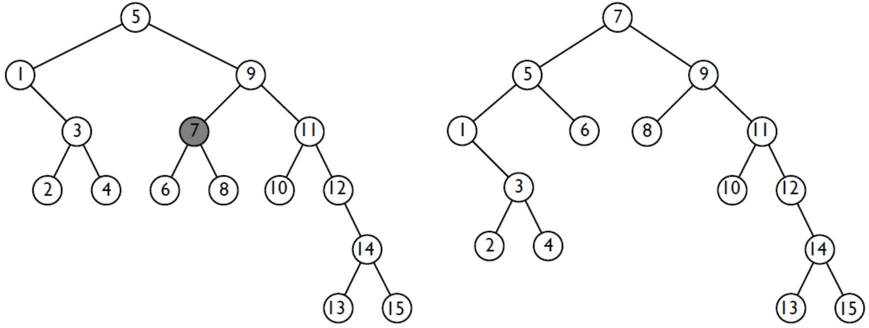


Demand-Aware

Fixed

Reconfigurable

$$BST_t \longrightarrow BST_{t+1}$$

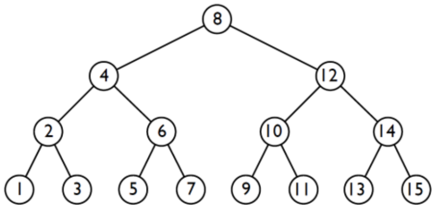


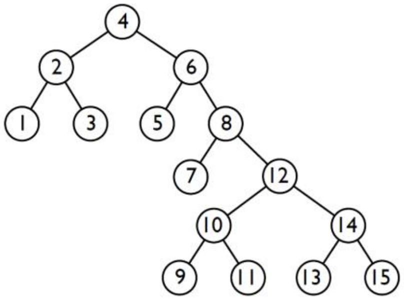
Datastructures

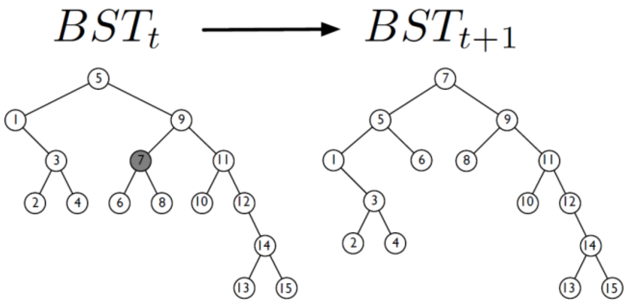
Oblivious

Demand-Aware

Self-Adjusting







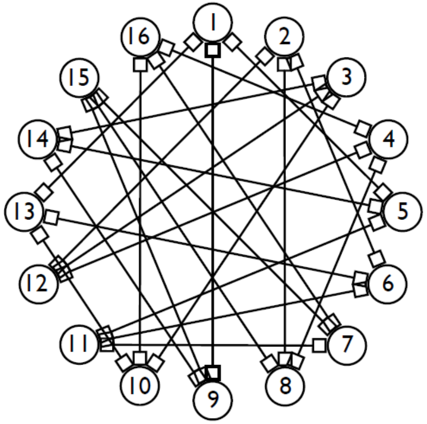
Lookup ***O(log n)***

Exploit **spatial locality**:
empirical entropy O(loglog n)

Exploit **temporal locality** as well:
O(1)

Analogously for Networks

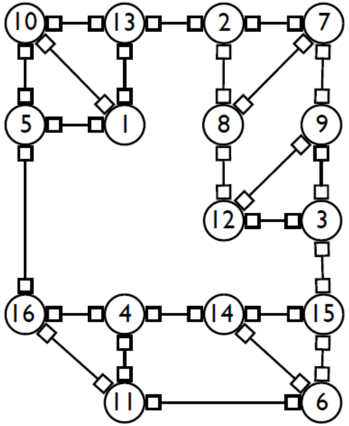
Oblivious



Const degree
(e.g., **expander**):
route lengths ***O(log n)***

▶▶

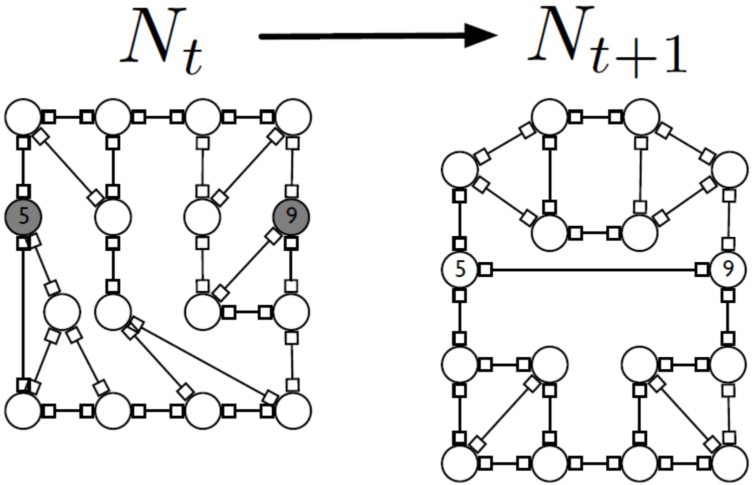
DAN



Exploit **spatial locality**: Route lengths depend on ***conditional entropy*** of demand

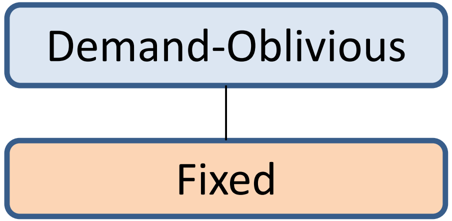
▶▶

SAN

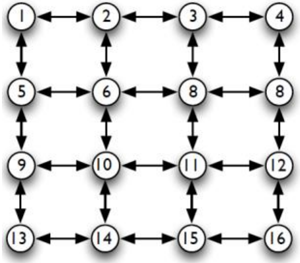


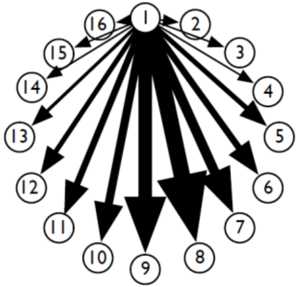
Exploit **temporal locality** as well

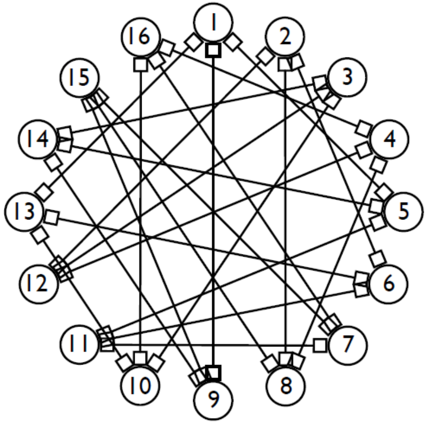
Oblivious Networks…



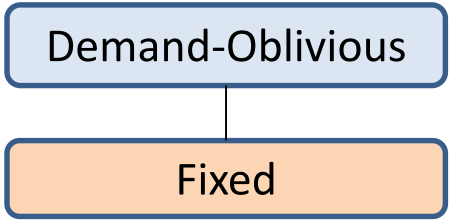
- Traditional, **fixed** networks (e.g. expander)
- Optimize for the **worst-case**
- Constant degree: communication partners at distance ***O(log n)*** from each other, **uniformly** and **independently** of their communication frequency
- Example **demands**:







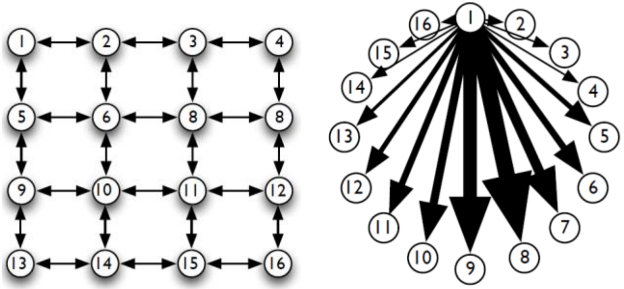
Oblivious Networks…

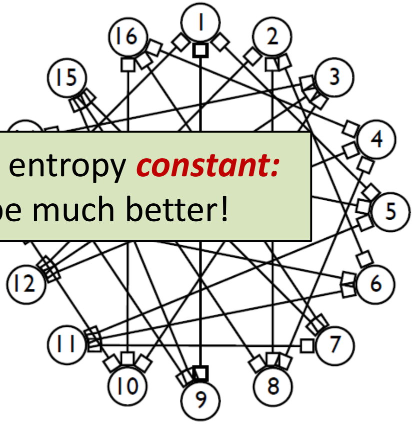


Demand-Oblivious

Fixed

- Traditional, **fixed** networks (e.g. expander)
- Optimize for the **worst-case**
- Constant degree: communication partners at distance ***O(log n)*** from each other, **uniformly** and **independently** of their communication frequency
- Example **demands**:





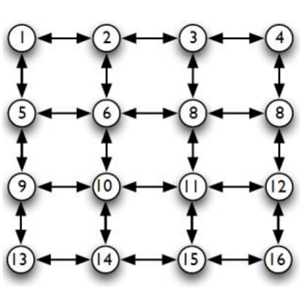
Conditional entropy ***constant:*** DANs would be much better!

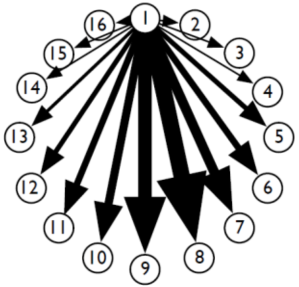
… DANs …

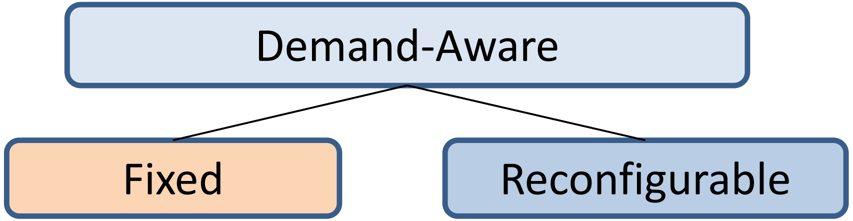
- **Demand-aware fixed** networks can take advantage of ***spatial locality***

- Optimize: place frequently communicating nodes close

- ***O(1)*** routes for our demands:



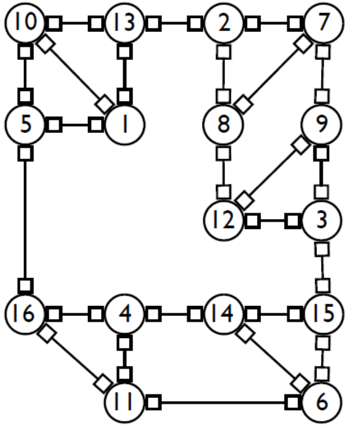




Demand-Aware

Fixed

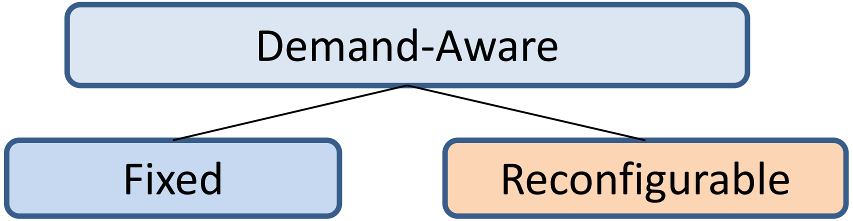
Reconfigurable



… SANs!

- **Demand-aware reconfigurable** networks can additionally take advantage of ***temporal locality***

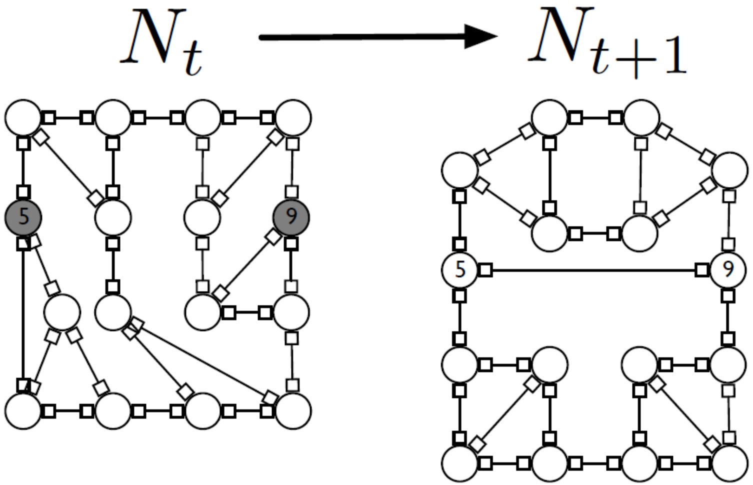
- By moving communicating elements close



Demand-Aware

Fixed

Reconfigurable



Diving a Bit Deeper: DAN

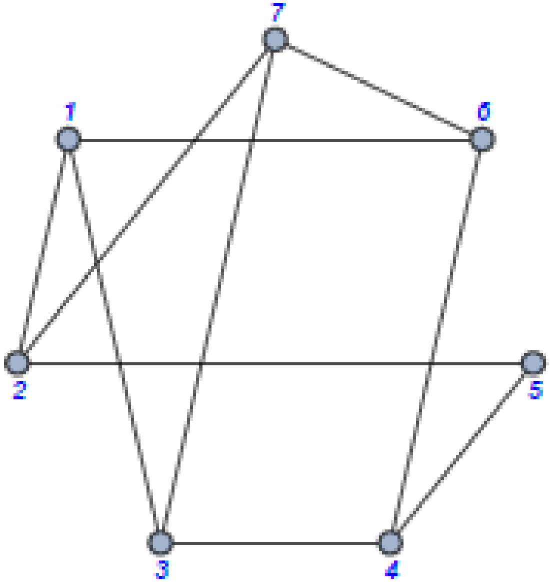
Workload: can be seen as graph as well.

Destinations

Sources

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}
$$

design



Demand matrix: joint distribution

DAN (of constant degree)

Diving a Bit Deeper: DAN

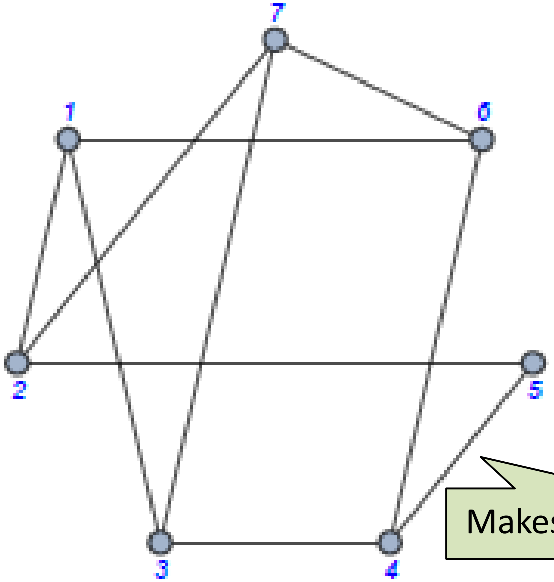
Destinations

Sources

	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{\text{[illegible]}}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	[illegible]	[illegible]	$\frac{\text{[illegible]}}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

Much from 4 to 5.

design



Makes sense to add link!

Demand matrix: joint distribution

DAN (of constant degree)

Diving a Bit Deeper: DAN

Destinations

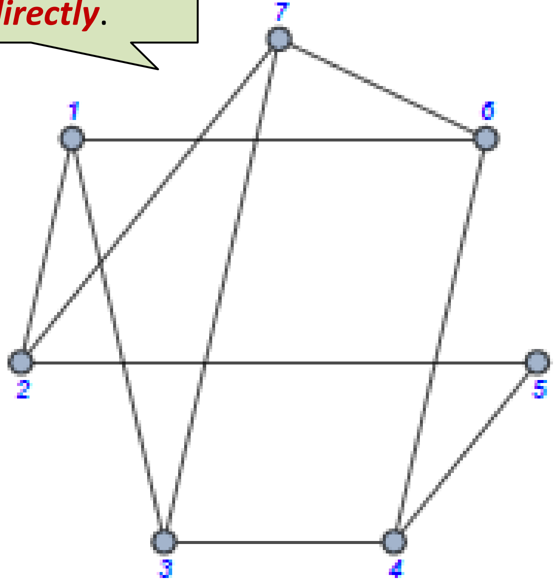
1 communicates to many.

Sources

						6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

design

Bounded degree: route to 7 ***indirectly***.



Demand matrix: joint distribution

DAN (of constant degree)

Diving a Bit Deeper: DAN

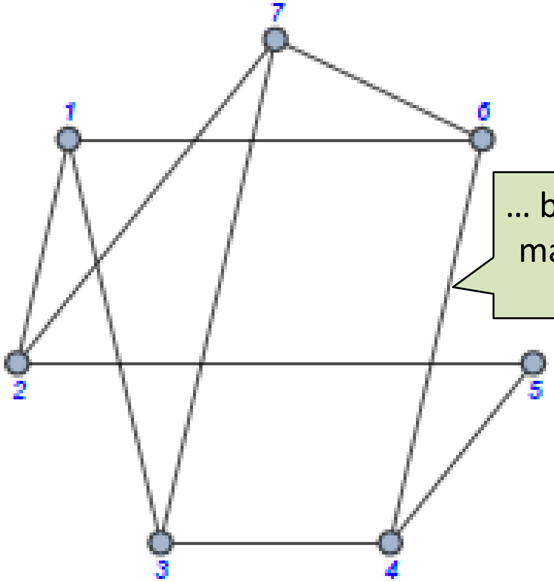
Destinations

Sources

$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & & \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & & \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}$$

4 and 6 don't communicate…

design

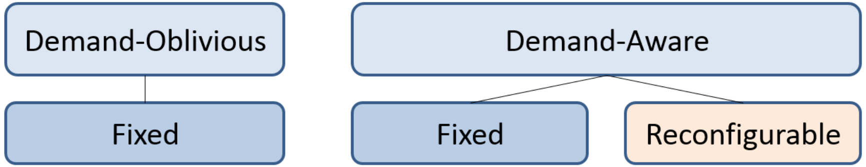


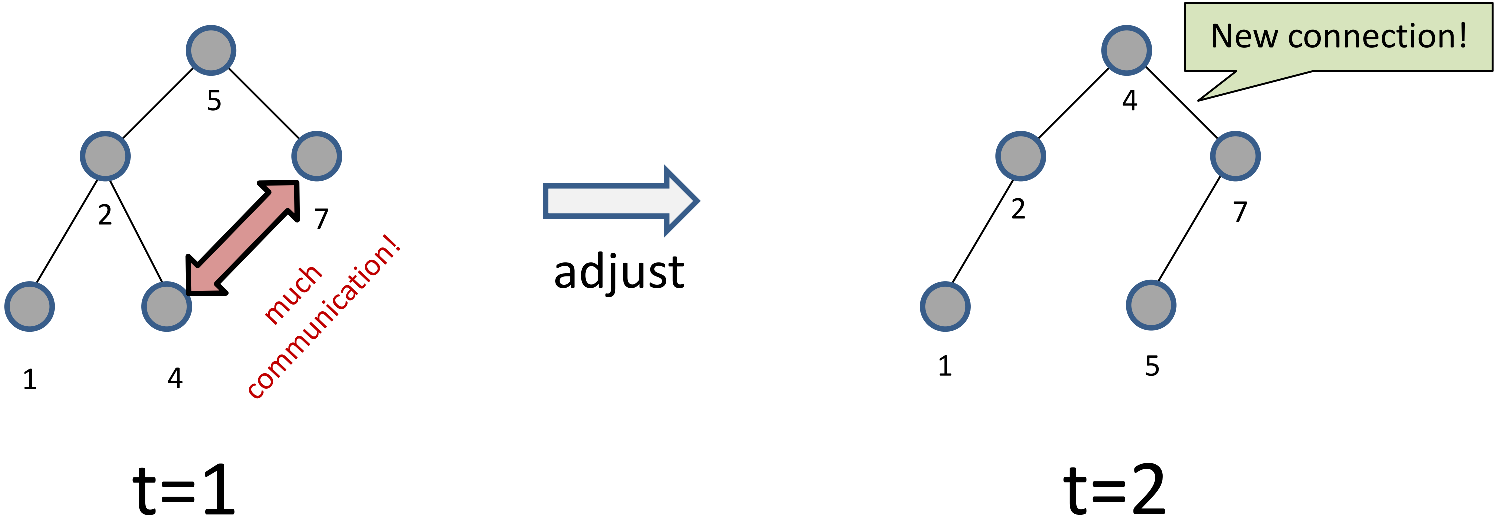
… but "extra" link still makes sense: ***not a subgraph***.

Demand matrix: joint distribution

DAN (of constant degree)

Example: Self-Adjusting Network (SANs) *Trees*





Challenges: How to **minimize reconfigurations**? How to keep network **locally** routable?

SplayNet: Towards Locally Self-Adjusting Networks. **TON** 2016.

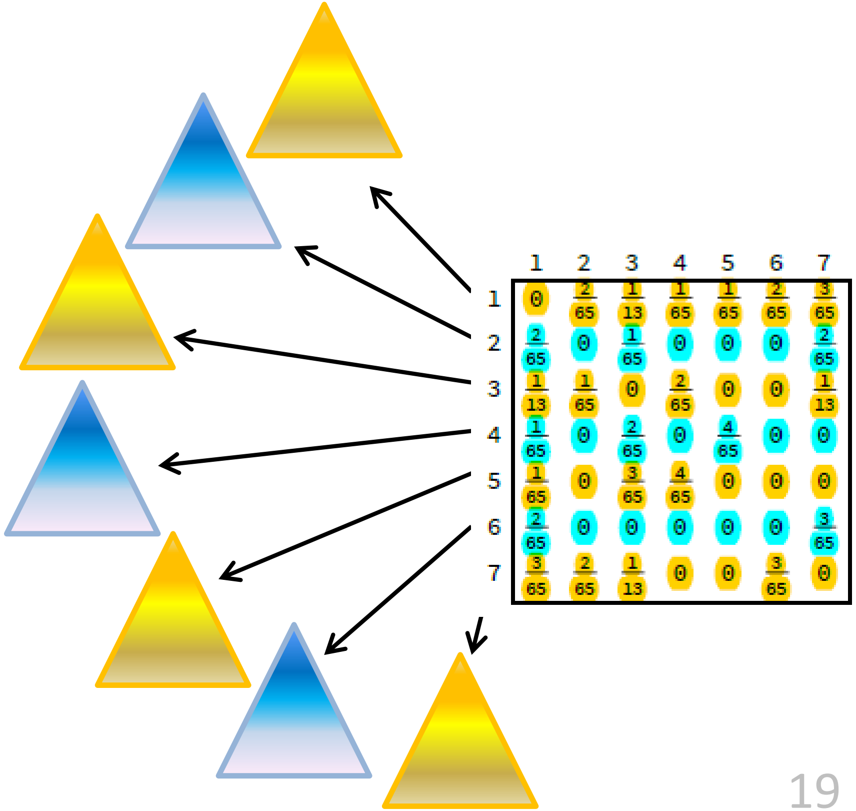
Lower Bound: Idea

- **Proof idea** (EPL=$\Omega(H_\Delta(Y|X))$):

- Build ***optimal*** Δ-ary tree for each source i: entropy lower bound known on EPL known for binary trees (***Mehlhorn*** 1975 for BST but proof does not need search property)

- Consider ***union*** of all trees

- Violates ***degree restriction*** but valid lower bound



	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

Lower Bound: Idea

Do this in **both dimensions**:

EPL ≥ $\Omega(\max\{H_\Delta(Y|X), H_\Delta(X|Y)\})$

$\Omega(H_\Delta(X|Y))$

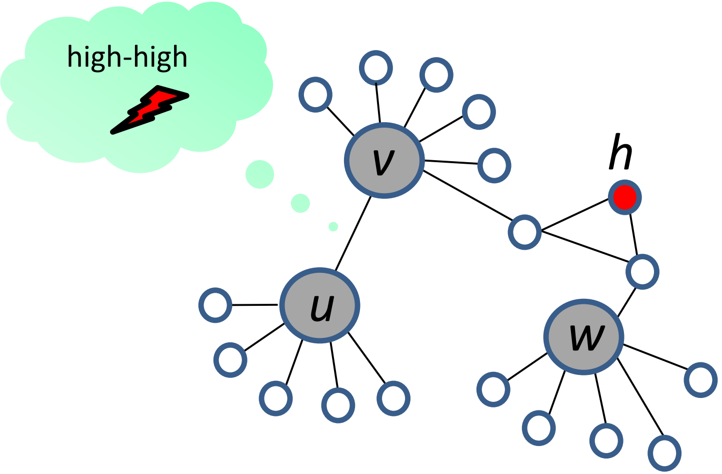
$\Omega(H_\Delta(Y|X))$

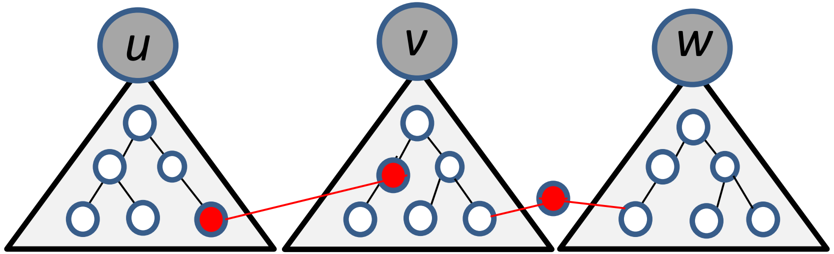
$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}$$

$\mathcal{D}$

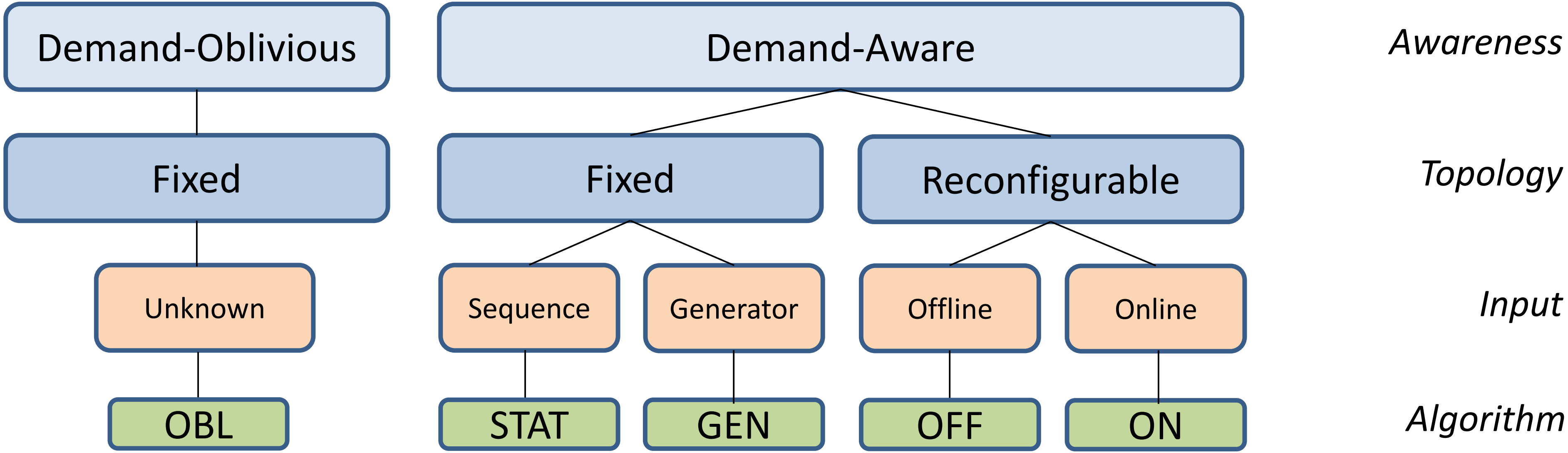
(Tight) Upper Bounds: Algorithm Idea

- Idea: construct **per-node optimal tree**
  - BST (e.g., Mehlhorn)
  - Huffman tree
  - Splay tree (!)
- Take **union** of trees but reduce degree
  - E.g., in sparse distribution: leverage **helper** nodes between two “large” (i.e., high-degree) nodes





Uncharted Space



Can compare to static or dynamic baseline!

Toward Demand-Aware Networking: A Theory for Self-Adjusting Networks. **ArXiv** 2018.

Managing Flexible Networks is Hard for Humans

Human Errors

Datacenter, enterprise, carrier networks: **mission-critical infrastructures**.
But even **techsavvy** companies struggle to provide reliable operations.

We discovered a misconfiguration on this pair of switches that caused what's called a "bridge loop" in the network.

A network change was […] executed incorrectly […] more "stuck" volumes and added more requests to the re-mirroring storm.

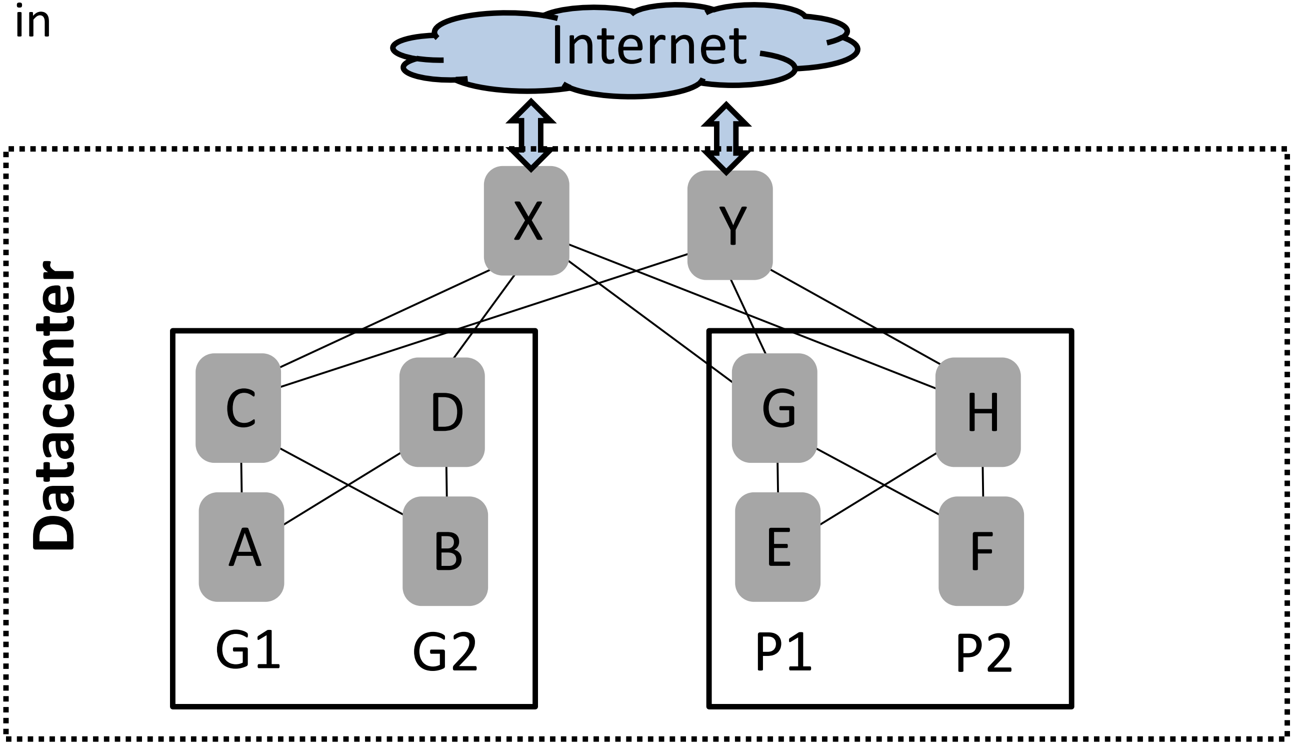
Service outage was due to a series of internal network events that corrupted router data tables.

Experienced a network connectivity issue […] interrupted the airline's flight departures, airport processing and reservations systems

Credits: Nate Foster

Example: Keeping Track of (Flexible) Routes Under Failures

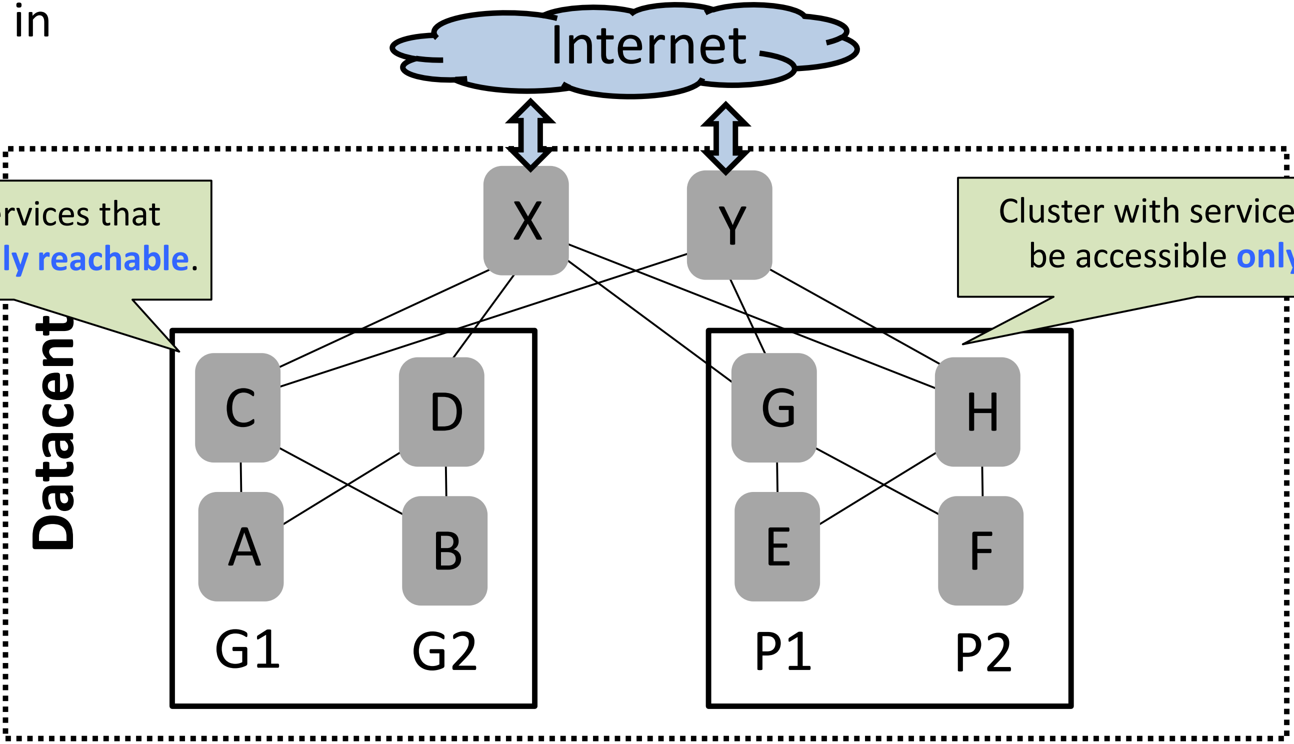
Example: BGP in
Datacenter



Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

Example: Keeping Track of (Flexible) Routes Under Failures

Example: BGP in **Datacenter**



Cluster with services that should be **globally reachable**.

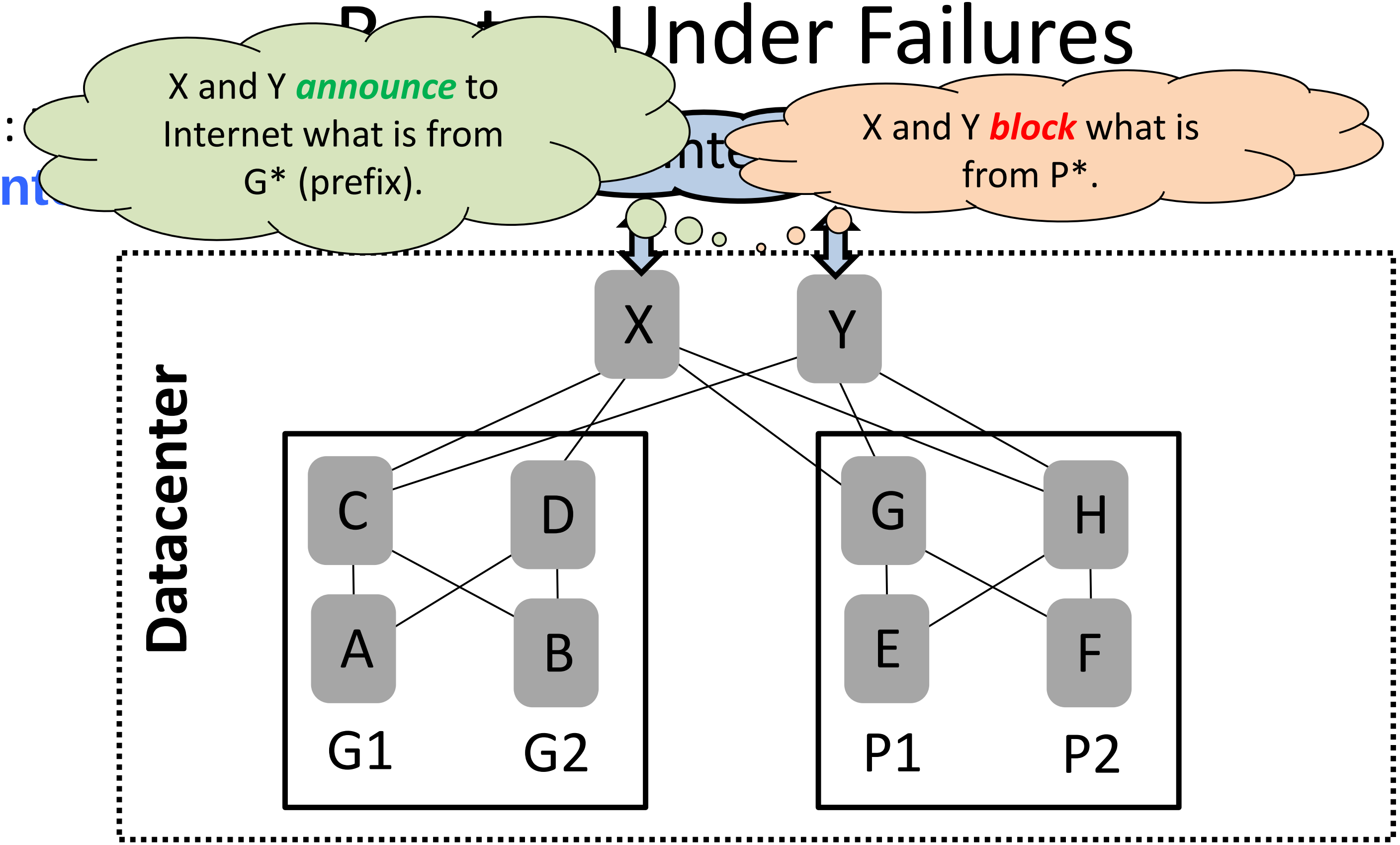
Cluster with services that should be accessible **only internally**.

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

Example: Keeping Track of (Flexible) R t Under Failures

Example:

Datacent


X and Y ***announce*** to Internet what is from G* (prefix).

X and Y ***block*** what is from P*.

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

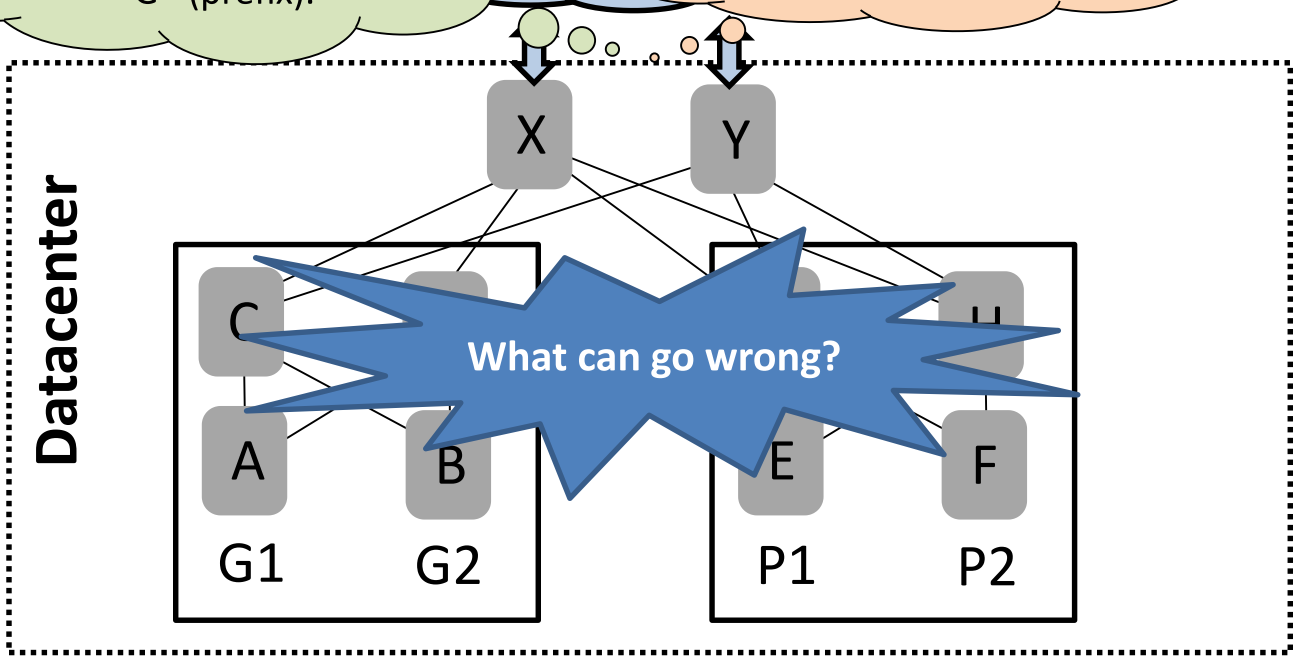
Example: Keeping Track of (Flexible) R t Under Failures

Example:

Datacent

X and Y ***announce*** to Internet what is from G* (prefix).

X and Y ***block*** what is from P*.



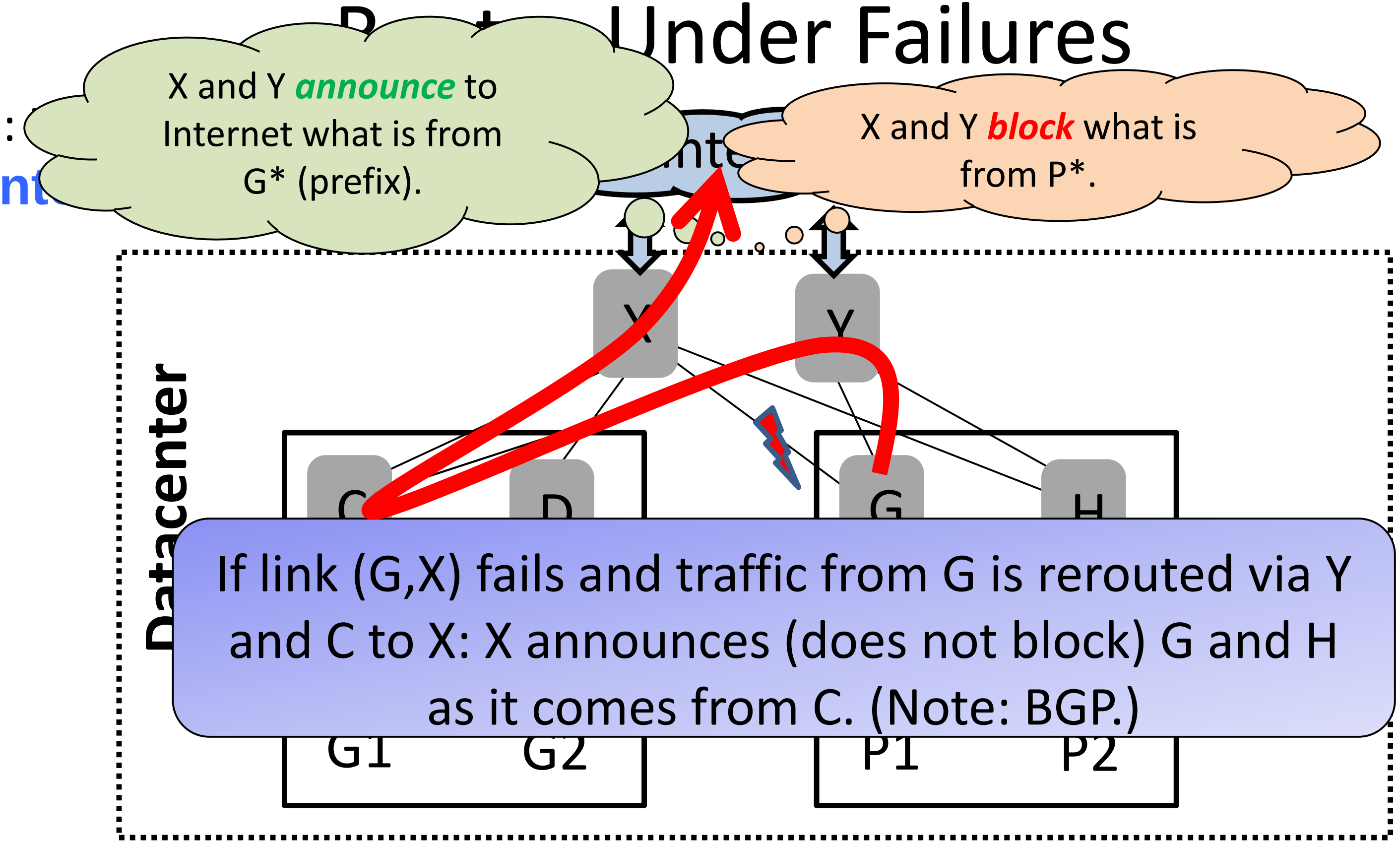
What can go wrong?

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

Example: Keeping Track of (Flexible) Routes Under Failures

Example:

Datacent



X and Y ***announce*** to Internet what is from G* (prefix).

X and Y ***block*** what is from P*.

If link (G,X) fails and traffic from G is rerouted via Y and C to X: X announces (does not block) G and H as it comes from C. (Note: BGP.)

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

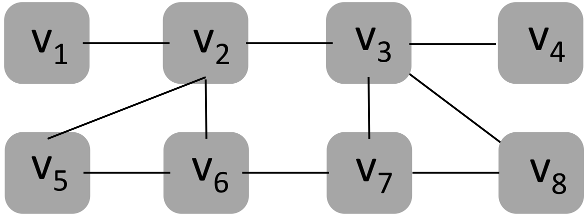
Managing Flexible Networks is Hard for Humans

The Case for Automation!

Role of Formal Methods?

Example: MPLS Networks

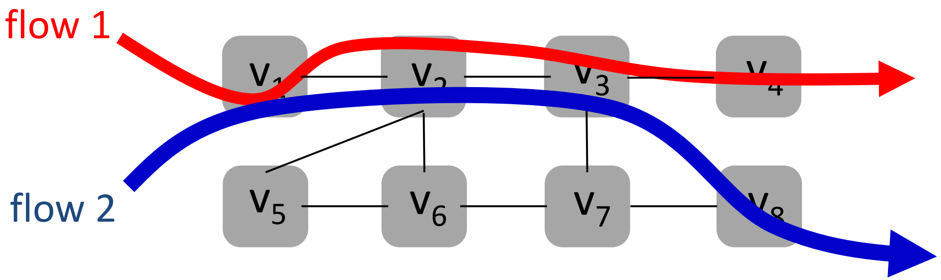
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

Example: MPLS Networks

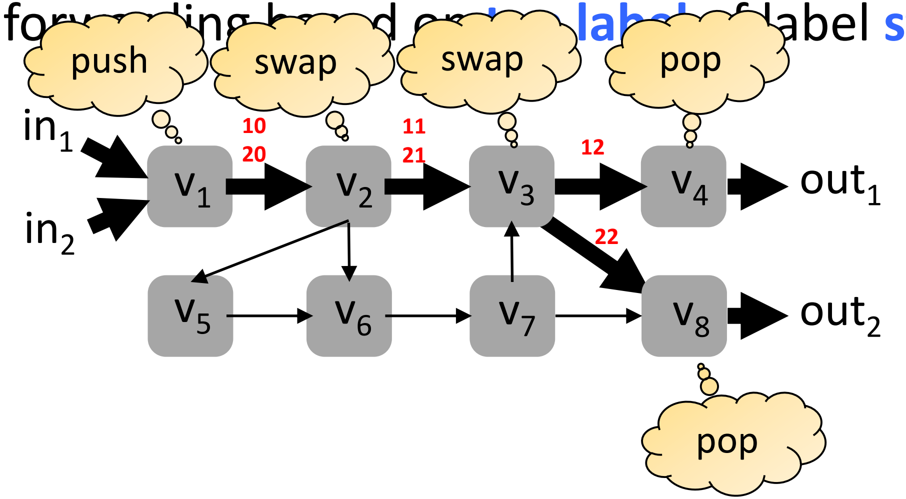
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

Example: MPLS Networks

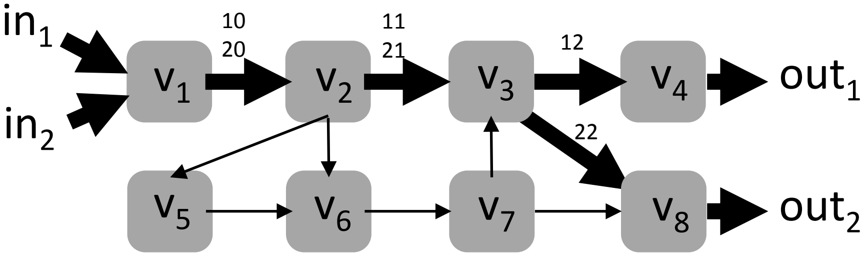
- MPLS: forwarding based on top label of label stack



Default routing of two flows

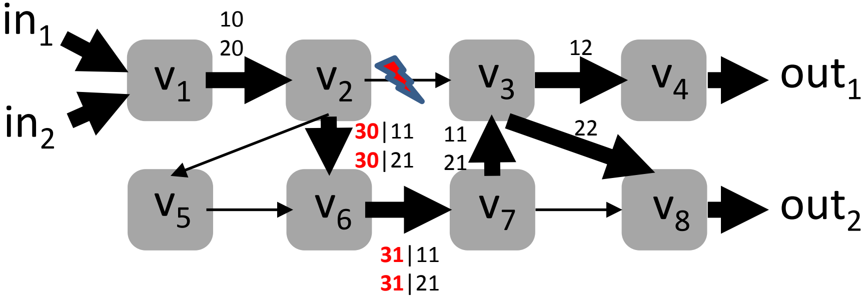
Fast Reroute Around *1 Failure*

- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

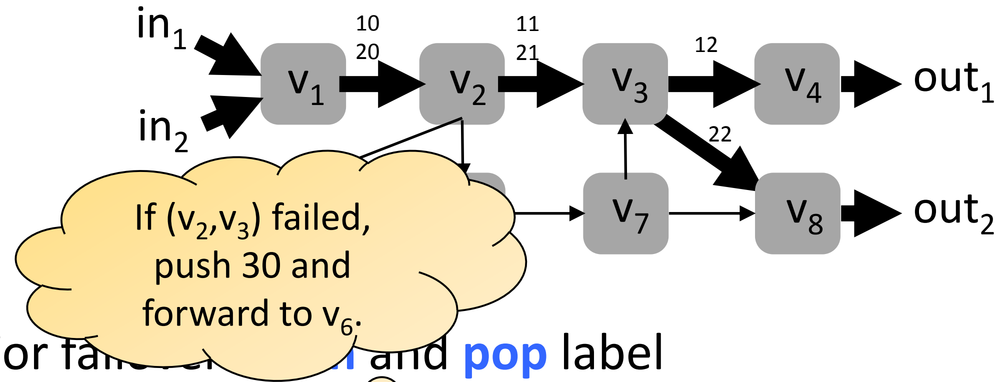
- For failover: **push** and **pop** label



One failure: push 30: route around (v_2,v_3)

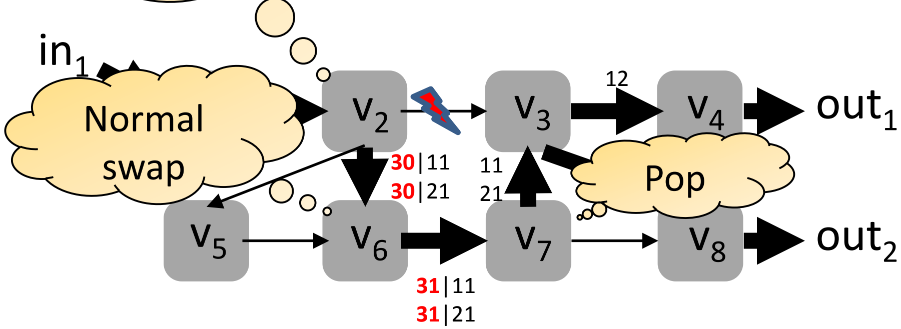
Fast Reroute Around *1 Failure*

- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

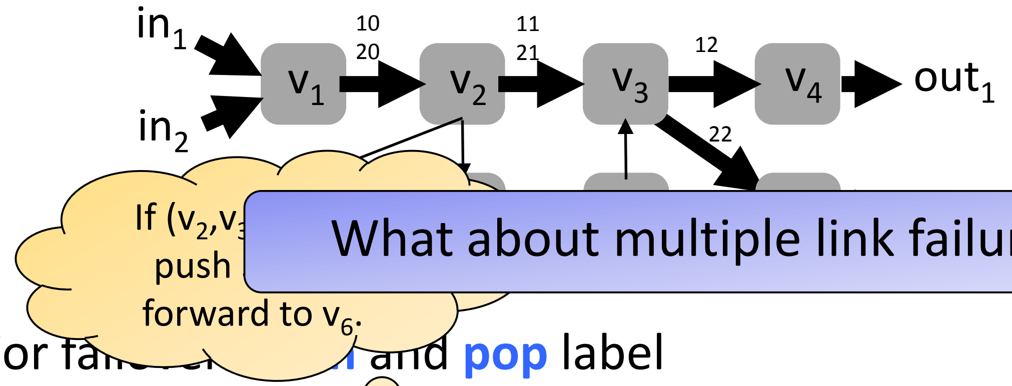
- For fa[illegible] and **pop** label



One failure: push 30: route around (v$_2$,v$_3$)

Fast Reroute Around *1 Failure*

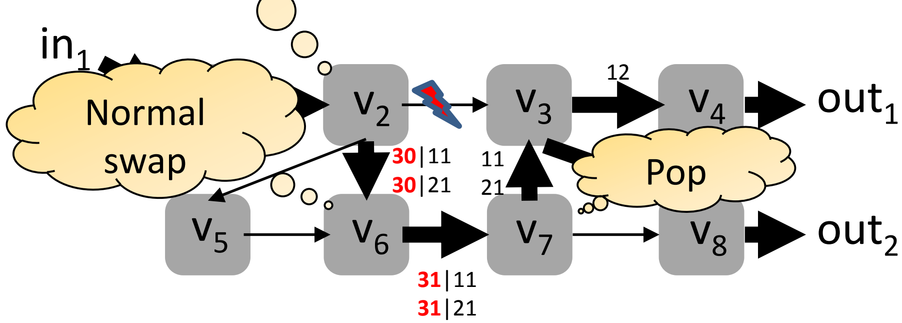
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

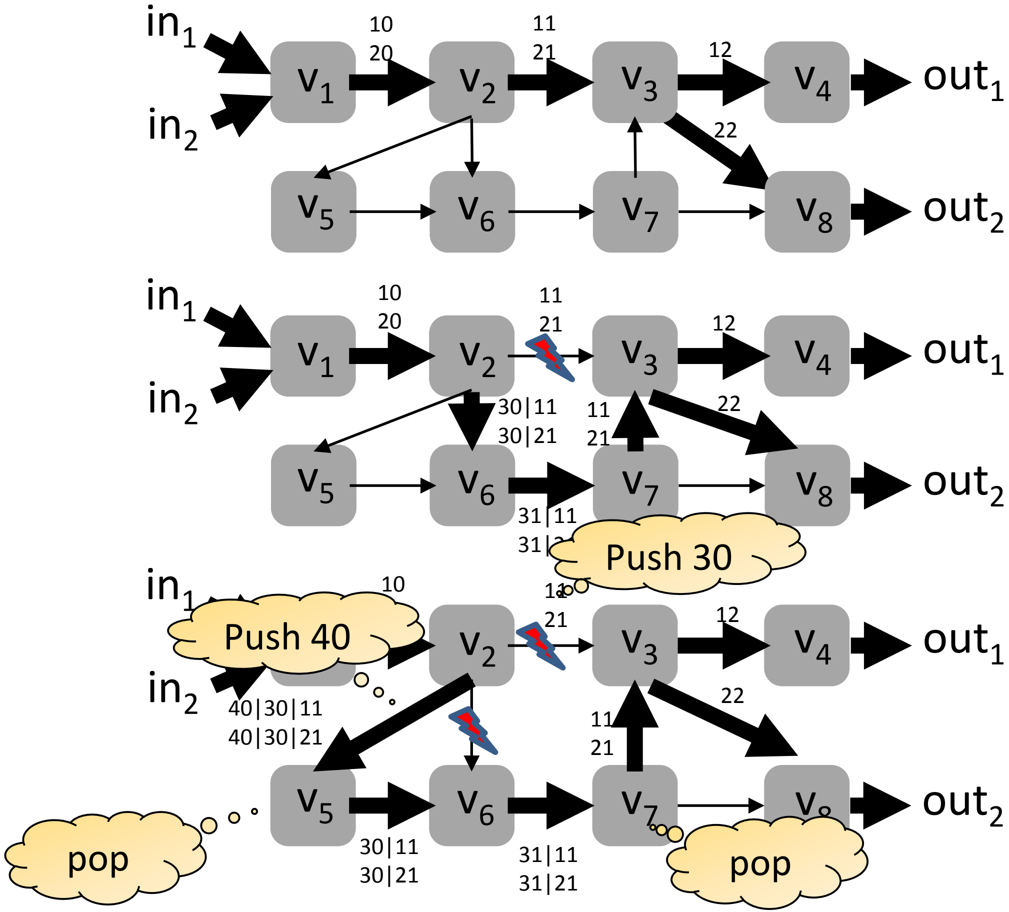
What about multiple link failures?

- For fa[illegible] and **pop** label



One failure: push 30: route around (v$_2$,v$_3$)

2 Failures: Push *Recursively*



Original Routing

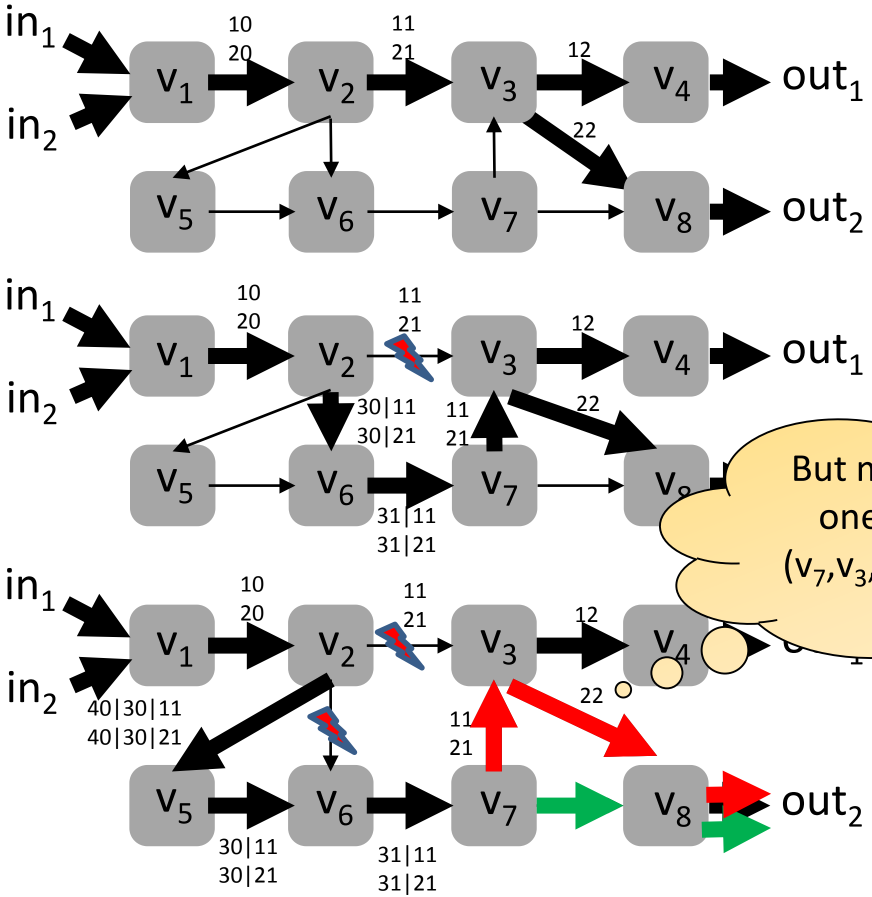
One failure: push 30: route around (v_2,v_3)

Two failures:

first push 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



Original Routing

One failure: push 30:

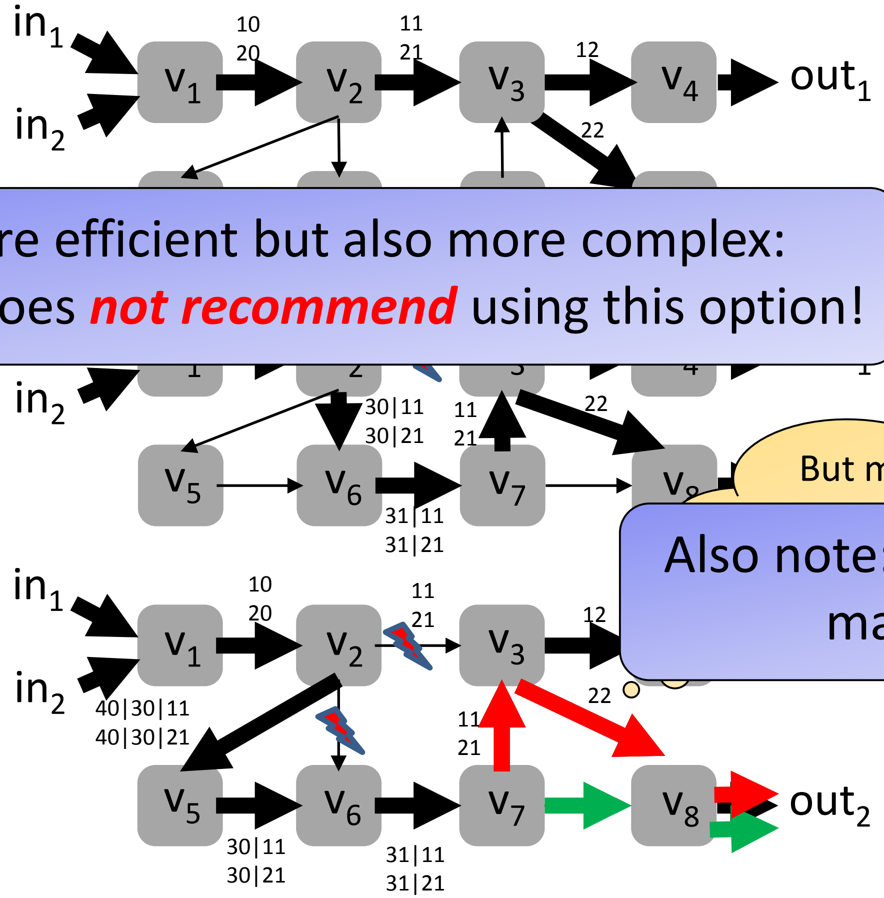
)

But masking links one-by-one can be inefficient: (v_7,v_3,v_8) could be shortcut to (v_7,v_8).

30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



Original Routing

More efficient but also more complex:
Cisco does ***not recommend*** using this option!

One failure: push 30:

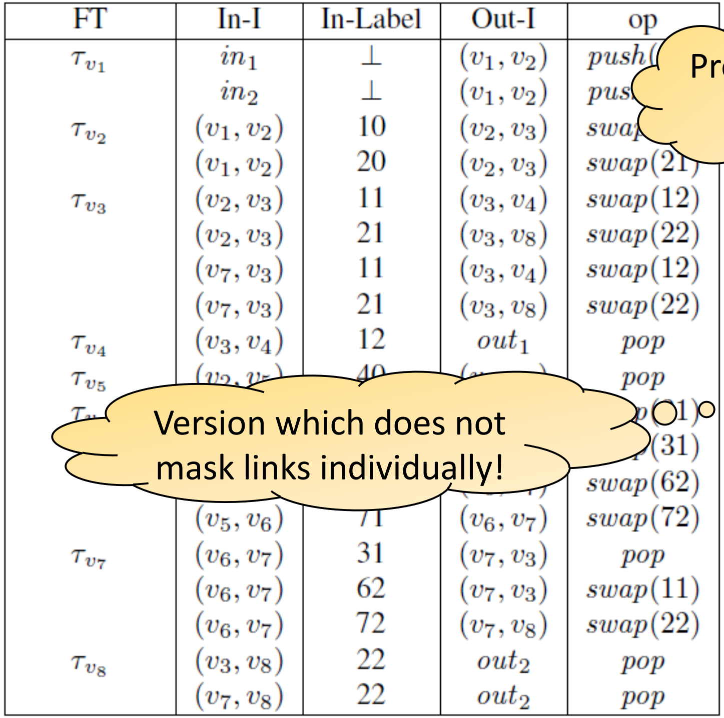
But masking links one-by-

Also note: due to push, ***header size*** may grow arbitrarily!

around (v_2,v_3)

Push recursively 40:
route around (v_2,v_6)

Forwarding Tables for Our Example



FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push($[illegible]
	in_2	$\perp$	(v_1, v_2)	pus[illegible]
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	swa[illegible]
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	[illegible]	pop
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	(31)
	[illegible]	[illegible]	[illegible]	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop

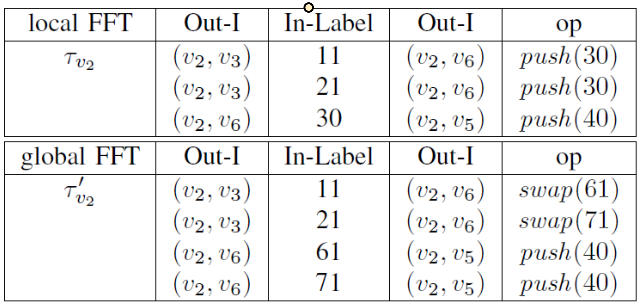
Version which does not mask links individually!

Flow Table

Protected link

Alternative link

Label



local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$

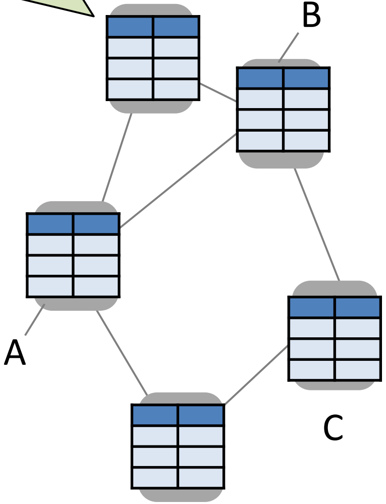
global FFT	Out-I	In-Label	Out-I	op
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

Failover Tables

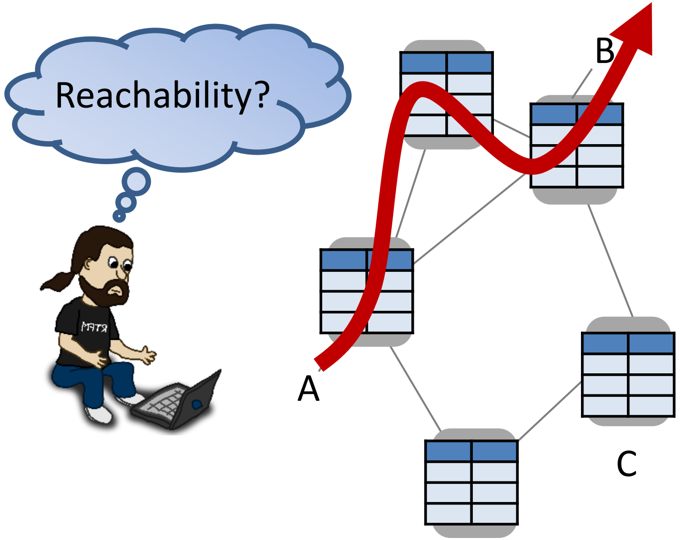
MPLS Tunnels in Today‘s ISP Networks

Responsibilities of a Sysadmin

Routers and switches store list of forwarding rules, and conditional failover rules.



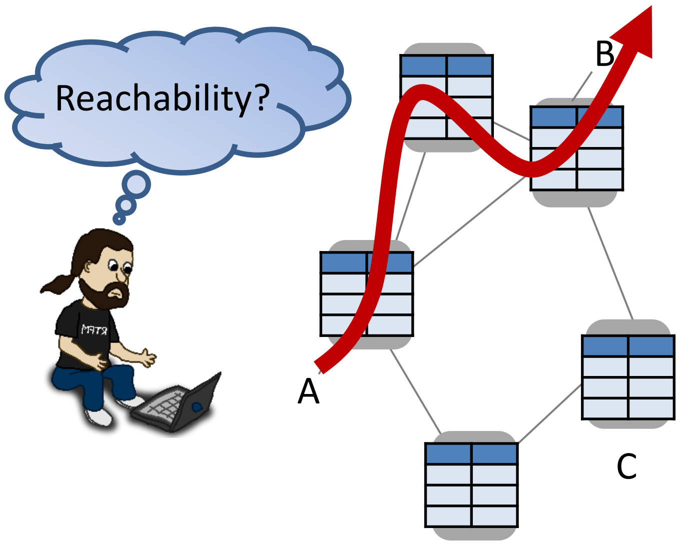
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?

Responsibilities of a Sysadmin

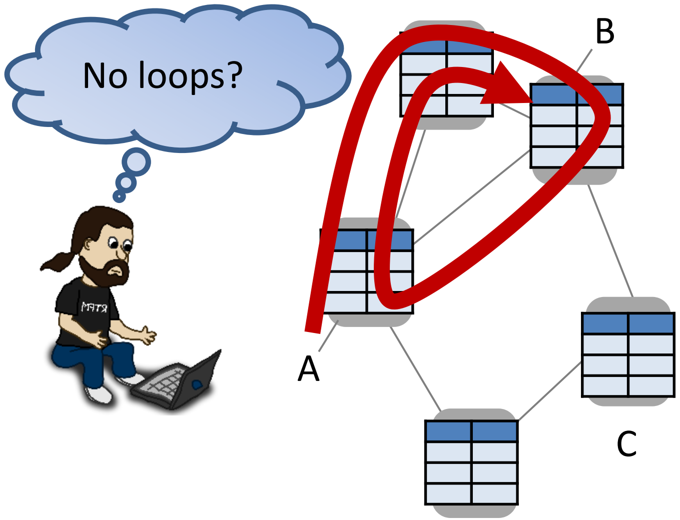


Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?

Or even more relevant for *QoS/QoE*: how long are detours?

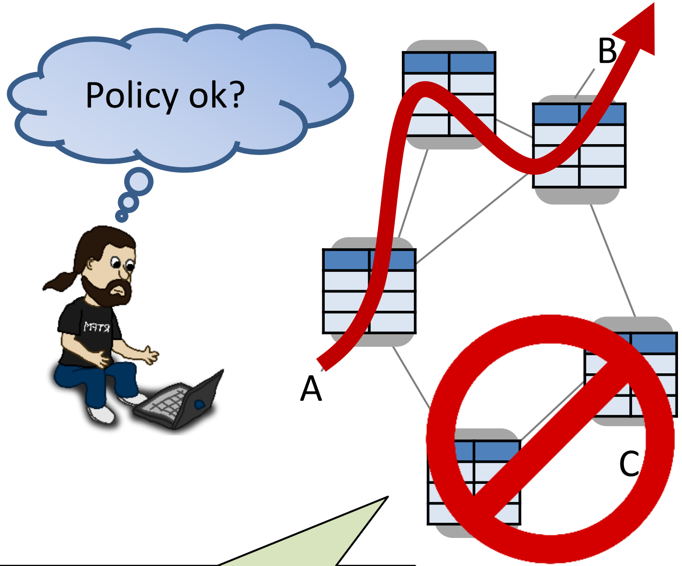
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?

Responsibilities of a Sysadmin

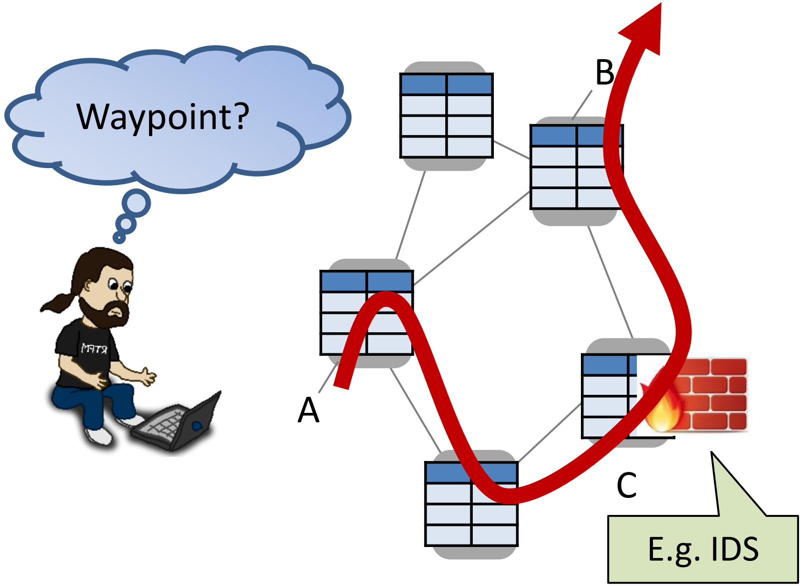


E.g. ***NORDUnet***: no traffic via Iceland (expensive!).

Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?

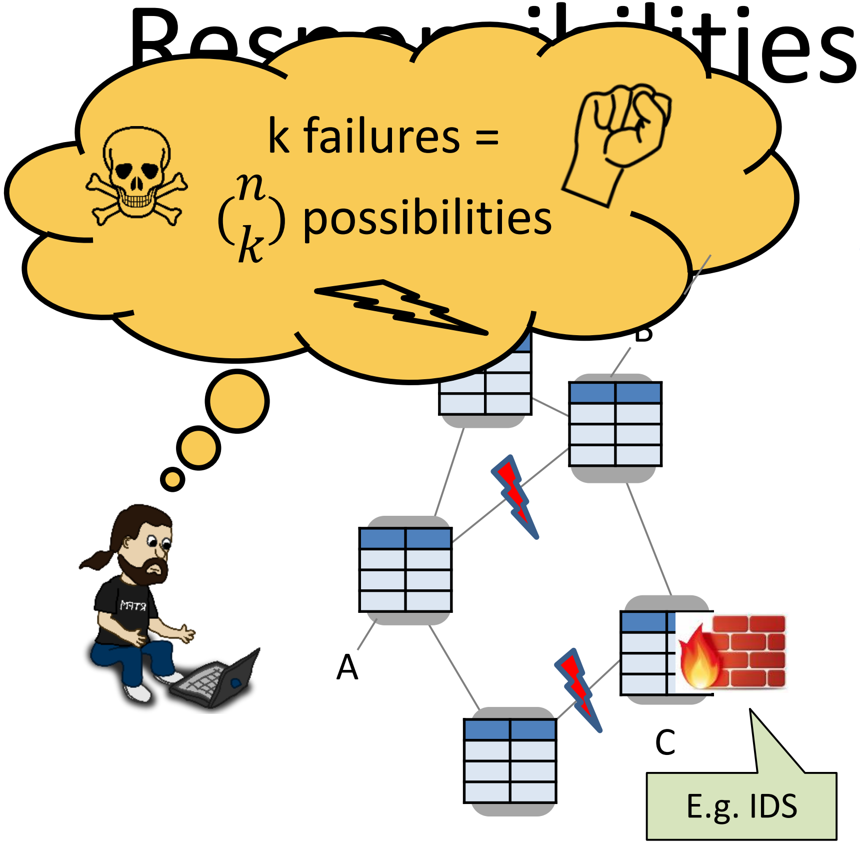
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or ***WAN optimizer***)?

Responsibilities of a Sysadmin



k failures = $\binom{n}{k}$ possibilities

<span style="color:blue">Sysadmin</span> responsible for:

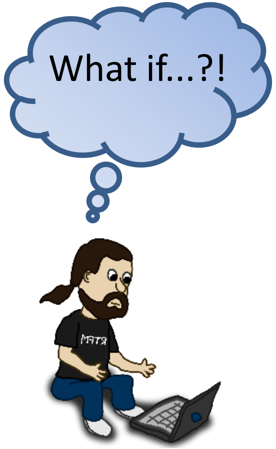
- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

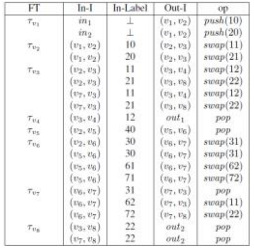
… and everything even under multiple failures?!

So what formal methods offer here?

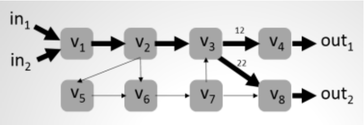
A lot!

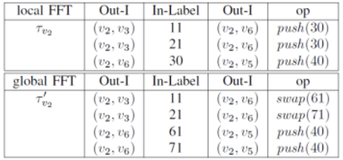
Leveraging Automata-Theoretic Approach





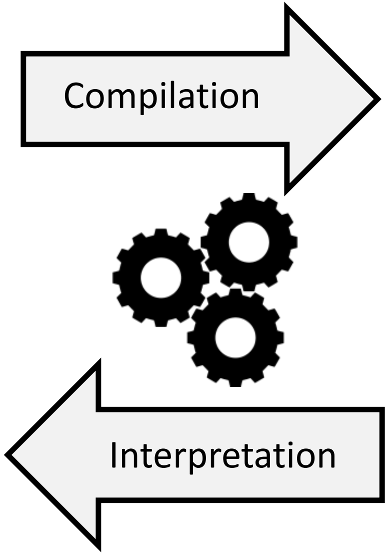
FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push(10)$
	in_2	$\perp$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop





local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$
global FFT	**Out-I**	**In-Label**	**Out-I**	**op**
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

MPLS **configurations**, Segment Routing etc.

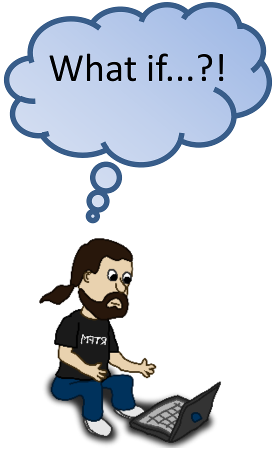


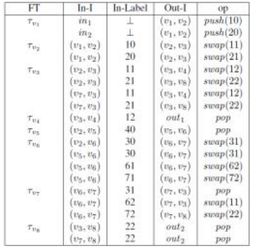
$$pX \Rightarrow qXX$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

Pushdown Automaton
and **Prefix Rewriting Systems** Theory

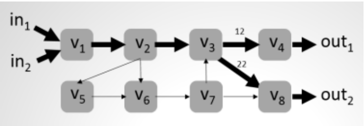
Leveraging Autom oach

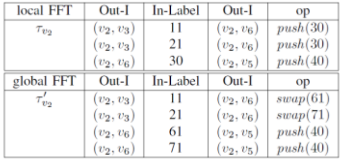
Use cases: Sysadmin ***issues queries*** to test certain properties, or do it on a ***regular basis*** automatically!





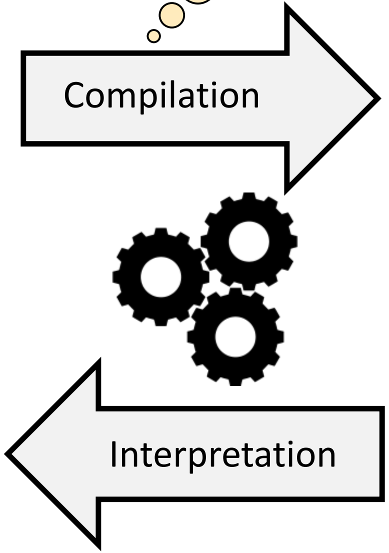
FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push(10)$
	in_2	$\perp$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop





local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$

global FFT	Out-I	In-Label	Out-I	op
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$


Compilation

Interpretation

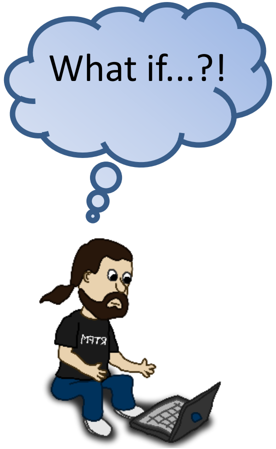
$$pX \Rightarrow qXX$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

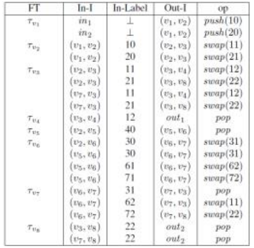
MPLS **configurations**, Segment Routing etc.

Pushdown Automaton and **Prefix Rewriting Systems** Theory

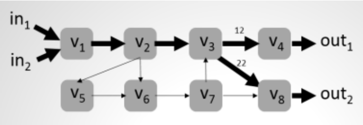
Leveraging Autom oach

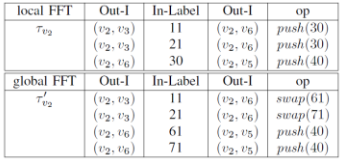
Use cases: Sysadmin ***issues queries*** to test certain properties, or do it on a ***regular basis*** automatically!



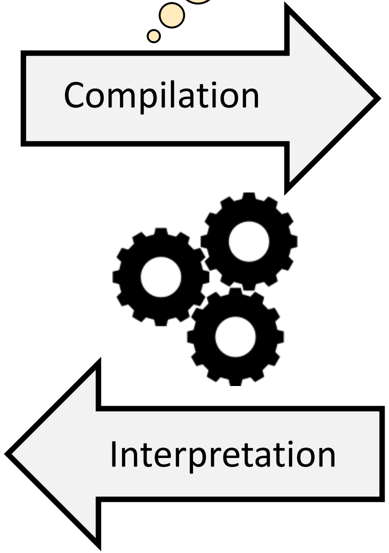


FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\bot$	(v_1, v_2)	$push(10)$
	in_2	$\bot$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop





local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$
global FFT	**Out-I**	**In-Label**	**Out-I**	**op**
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$



$$\mathrm{pX} \Rightarrow \mathrm{qXX}$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

MPLS **configurations**,

c.

Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks. **INFOCOM** 2018.

Pushdown Automaton and **Prefix Rewriting Systems** Theory

Mini-Tutorial: A Network Model

- Network: a 7-tuple

Links

Outgoing interfaces

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Nodes

Incoming interfaces

Set of labels in packet header

Mini-Tutorial: A Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Interface function

Interface function: maps outgoing interface to next hop node and incoming interface to previous hop node

$$\lambda_v : I_v^{in} \cup I_v^{out} \rightarrow V$$

That is: $(\lambda_v(in), v) \in E$ and $(v, \lambda_v(out)) \in E$

Mini-Tutorial: A Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Routing function

Routing function: for each set of failed links $F \subseteq E$, the routing function

$$\delta_v^F : I_v^{in} \times L^* \rightarrow 2^{(I^{out} \times L^*)}$$

defines, for all incoming interfaces and packet headers, outgoing interfaces together with modified headers.

Routing in Network

Packet routing sequence can be represented using sequence of tuples:

$$(v_i, in_i, h_i, out_i, h_{i+1}, F_i)$$

Node receives...

... on interface...

... packet with header...

... forwards it to live next hop...

... with new header..

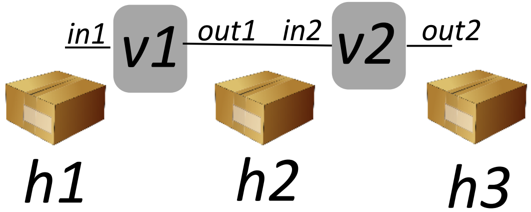
... given that these links are down.

- Example: **routing** (in)finite sequence of tuples

$$(v_1, in_1, h_1, out_1, h_2, F_1),$$

$$(v_2, in_2, h_2, out_2, h_3, F_2),$$

$$\ldots$$



Example Rules:
Regular Forwarding on Top-Most Label

Push:

Push label on stack

$$(v, in)\ell \rightarrow (v, out, 0)\ell'\ell \text{ if } \tau_v(in, \ell) = (out, push(\ell'))$$

Swap:

Swap top of stack

$$(v, in)\ell \rightarrow (v, out, 0)\ell' \text{ if } \tau_v(in, \ell) = (out, swap(\ell'))$$

Pop:

Pop top of stack

$$(v, in)\ell \rightarrow (v, out, 0) \text{ if } \tau_v(in, \ell) = (out, pop)$$

Example *Failover* Rules

Emumerate all rerouting options

Failover-Push:

$$(v, out, i)\ell \to (v, out', i+1)\ell'\ell \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', push(\ell'))$$

Failover-Swap:

$$(v, out, i)\ell \to (v, out', i+1)\ell' \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', swap(\ell')),$$

Failover-Pop:

$$(v, out, i)\ell \to (v, out', i+1) \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', pop).$$

Example rewriting sequence:

$$(v_1, in_1)h_1\bot \to (v_1, out, 0)h\bot \to (v_1, out', 1)h'\bot \to (v_1, out'', 2)h''\bot \to \dots \to (v_1, out_1, i)h_2\bot$$

Try default

Try first backup

Try second backup

A Complex and Big Formal Language! Why Polynomial Time?!

- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

A Complex and Big Formal Language!
Why Polynomial Time?!

- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!
- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

This is ***not*** how we will use the PDA!

A Complex and Big Formal Language!
Why Polynomial Time?!

k failures = $\binom{n}{k}$ possibilities

- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!
- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

The words in our language are sequences of pushdown stack symbols, not the labels of transitions.

Time for Automata Theory!

- Classic result by **Büchi** 1964: the set of all reachable configurations of a pushdown automaton a is regular set

- Hence, we can operate only on Nondeterministic Finite Automata (NFAs) when reasoning about the pushdown automata

- The resulting **regular operations** are all polynomial time

- Important result of **model checking**

Julius Richard Büchi

1924-1984

Swiss logician

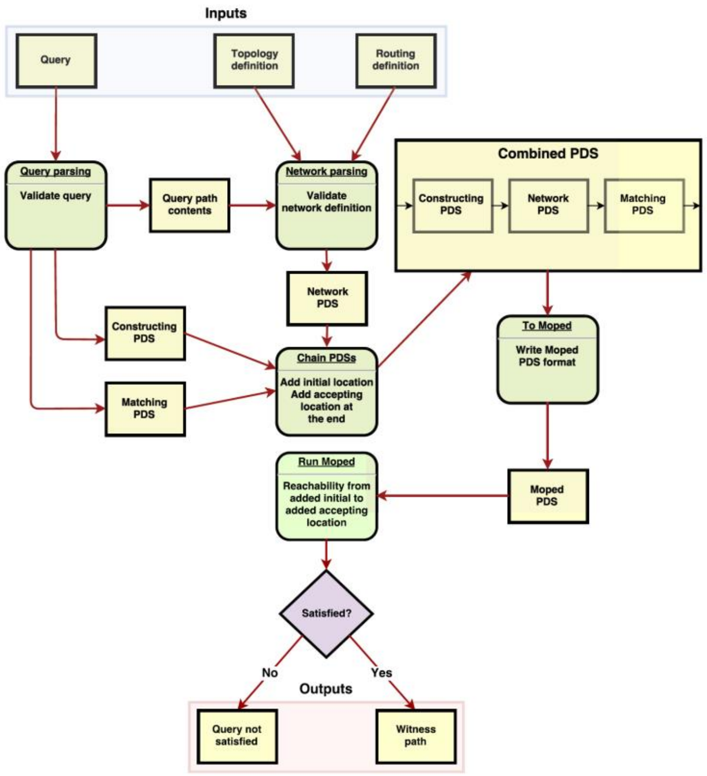
Preliminary Tool and Query Language

Part 1: Parses query and constructs Push-Down System (PDS)

- In Python 3

Part 2: Reachability analysis of constructed PDS

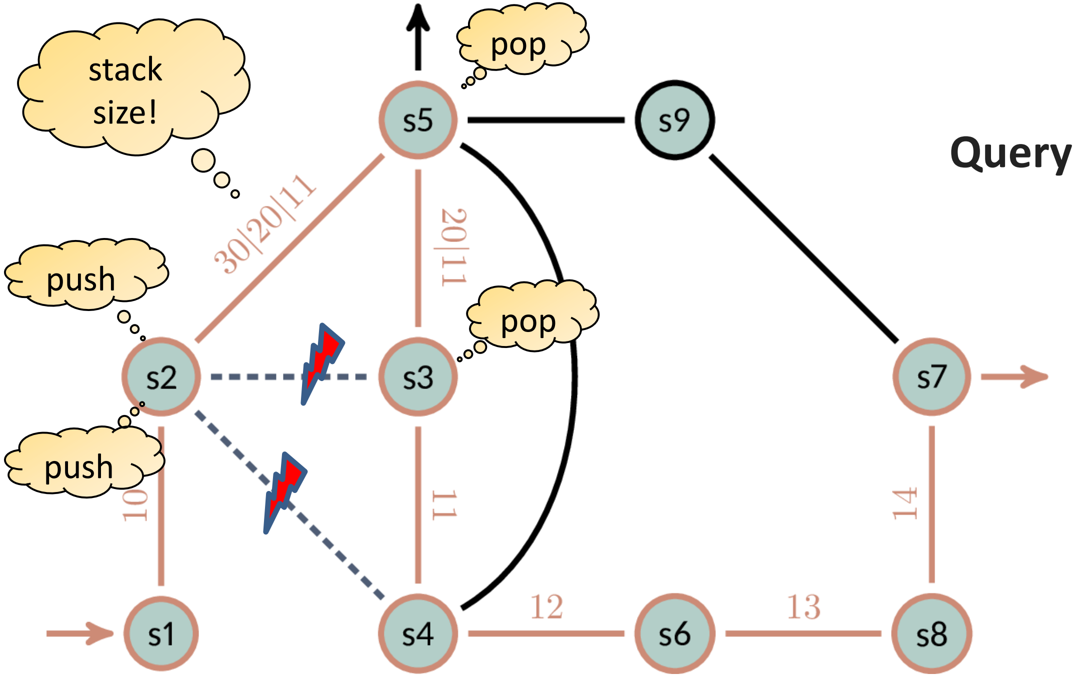
- Using ***Moped*** tool



query processing flow

Example: Traversal Testing With 2 Failures

Traversal test with k=2: Can traffic starting with [] go through s5, under up to k=2 failures?



2 failures

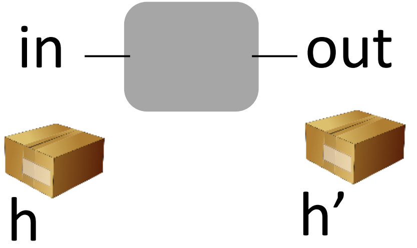
Query: k=2 [] s1 >> s5 >> s7 []

YES!

(Gives witness!)

But What About Other Networks?!

The **clue**: exploit the specific structure of MPLS rules.



Rules match the header **h** of packets arriving at ***in***, and define to which port ***out*** to forward as well as new header **h’**.

Rules of general networks (e.g., SDN):

arbitrary header rewriting

$$in \times L^* \rightarrow out \times L^*$$

VS

(Simplified) MPLS rules:

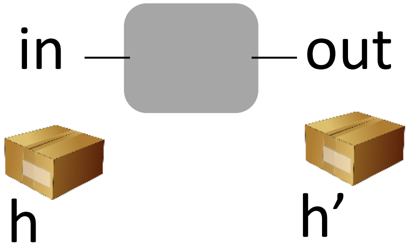
prefix rewriting

$$in \times L \rightarrow out \times \boldsymbol{OP}$$

where *OP* = {*swap,push,pop*}

But What About Other Networks?!

The **clue**: exploit the specific structure of MPLS rules.



Rules match the header **h** of packets arriving at ***in***, and define to which port ***out*** to forward as well as new header **h’**.

Rules of general netwo g., SDN):

arbitrary hea ewriting

*in x L → out x L**

Undecidable!

VS

(Simplified) MP s:

prefix r ing

→ *out x* ***OP***

wh OP = {swap,push,pop}

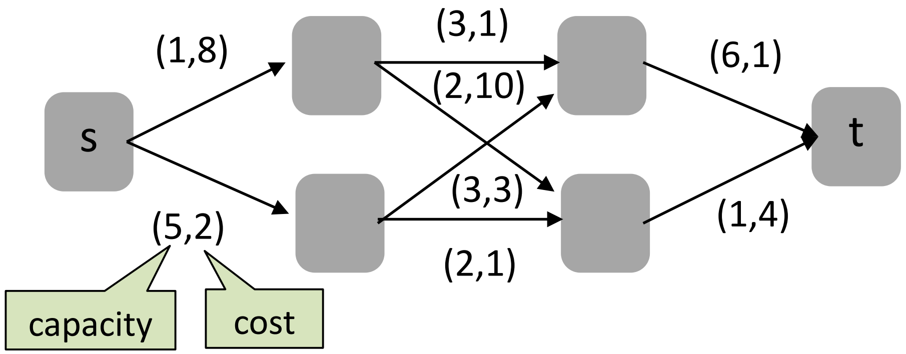
Polynomial-time!

What about QoS and Quantitative Aspects?

First Approaches: WNetKAT

- A **weighted** SDN programming and verification language

- Goes ***beyond topological*** aspects but account for:
  - actual **resource** availabilities, **capacities**, **costs**, or even **stateful** operations

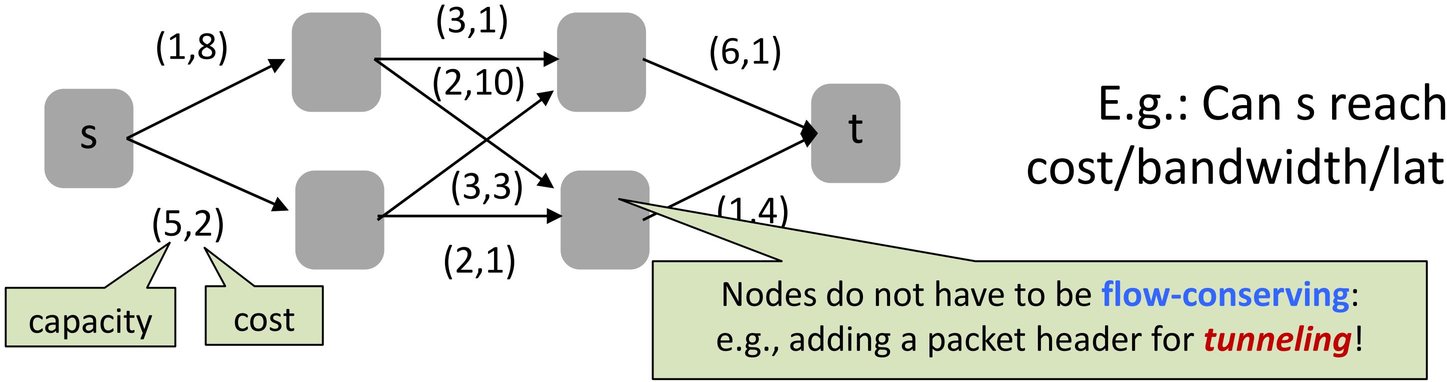


E.g.: Can s reach t at cost/bandwidth/latency x?

First Approaches: WNetKAT

- A **weighted** SDN programming and verification language

- Goes ***beyond topological*** aspects but account for:
  - actual **resource** availabilities, **capacities**, **costs**, or even **stateful** operations



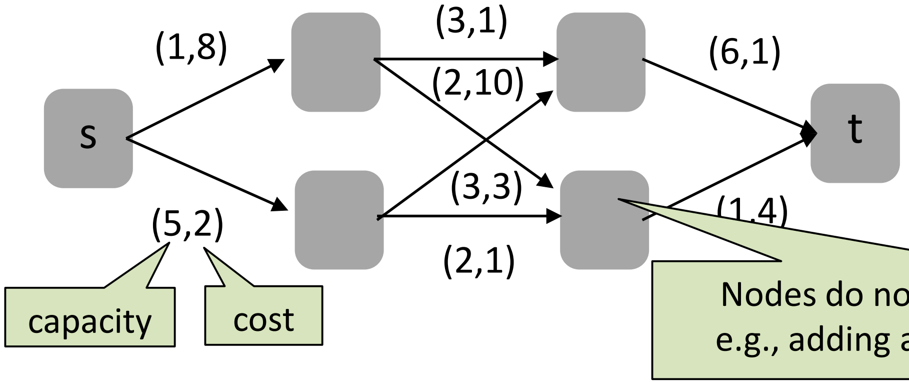
Nodes do not have to be **flow-conserving**: e.g., adding a packet header for ***tunneling***!

E.g.: Can s reach t at cost/bandwidth/latency x?

First Approaches: WNetKAT

- A **weighted** SDN programming and verification language

- Goes ***beyond topological*** aspects but account for:
  - actual **resource** availabilities, **capacities**, **costs**, or even **stateful** operations



E.g.: Can s reach t at cost/bandwidth/latency x?

Nodes do not have to be **flow-conserving**:
e.g., adding a packet header for ***tunne***

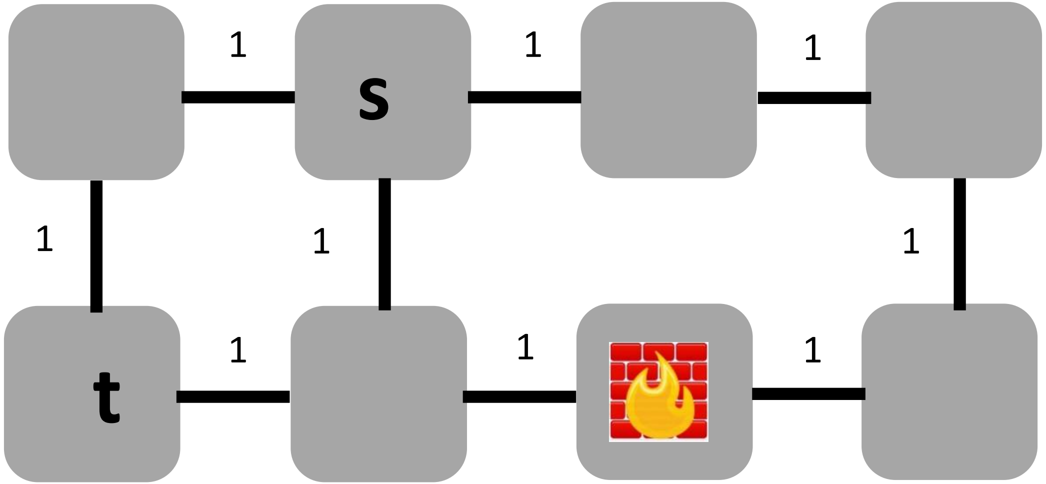
WNetKAT. ***OPODIS*** 2016.

In General: Exploiting Flexibilities is Even Hard for Computers

Part A: Because Algorithmic Problems are (***Computationally***) Complex

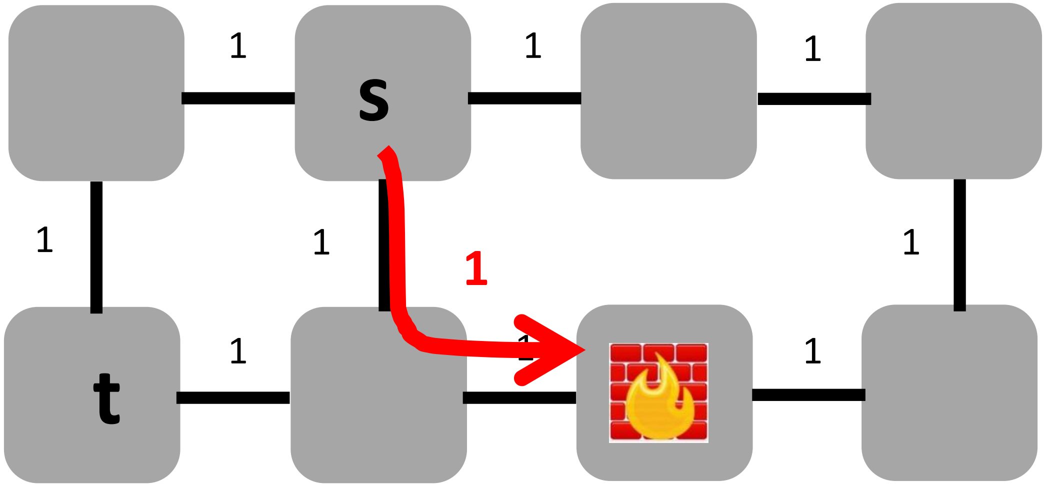
(Waypoint-)Routing is Hard

- Routing through a waypoint



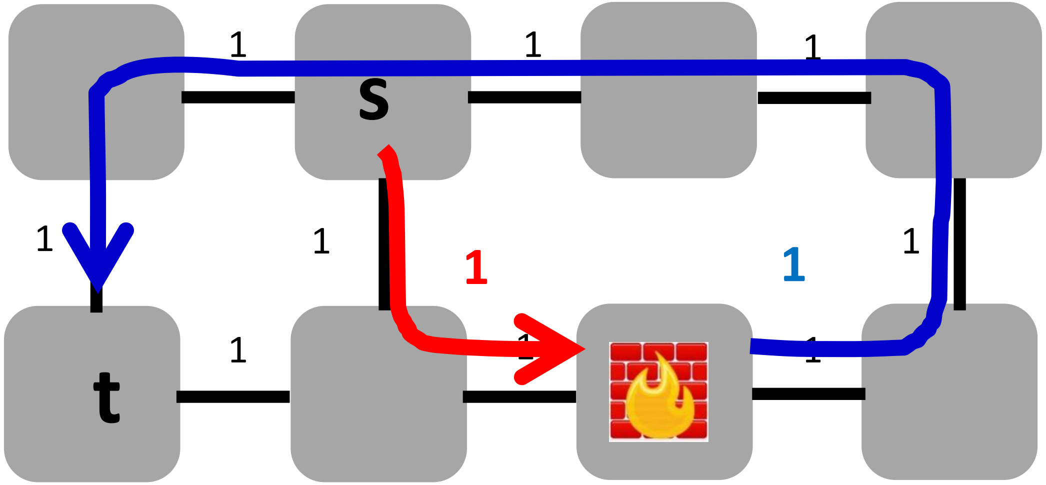
(Waypoint-)Routing is Hard

- Routing through a waypoint
- Greedy fails…



(Waypoint-)Routing is Hard

- Routing through a waypoint
- Greedy fails…

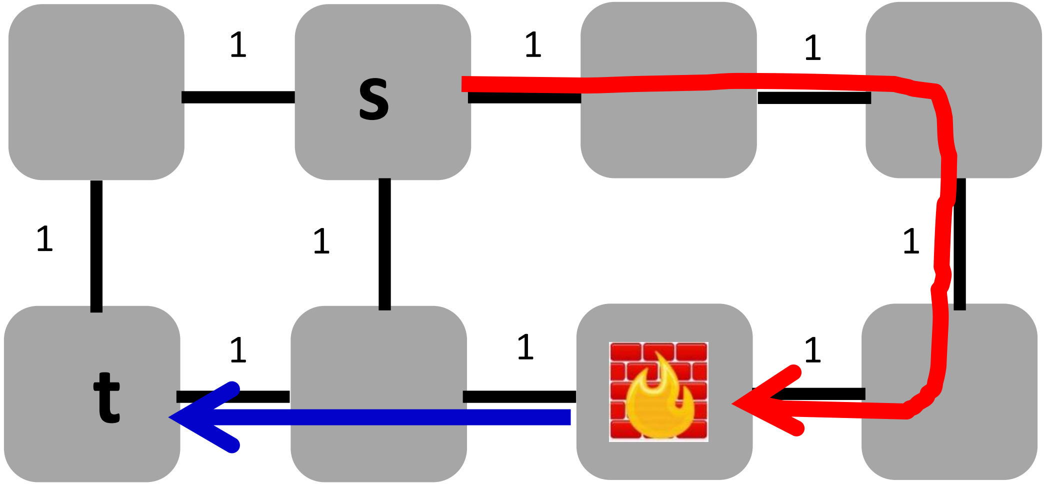


Total length: 2+6=8

(Waypoint-)Routing is Hard

- Routing through a waypoint

- Greedy fails…



Total length:

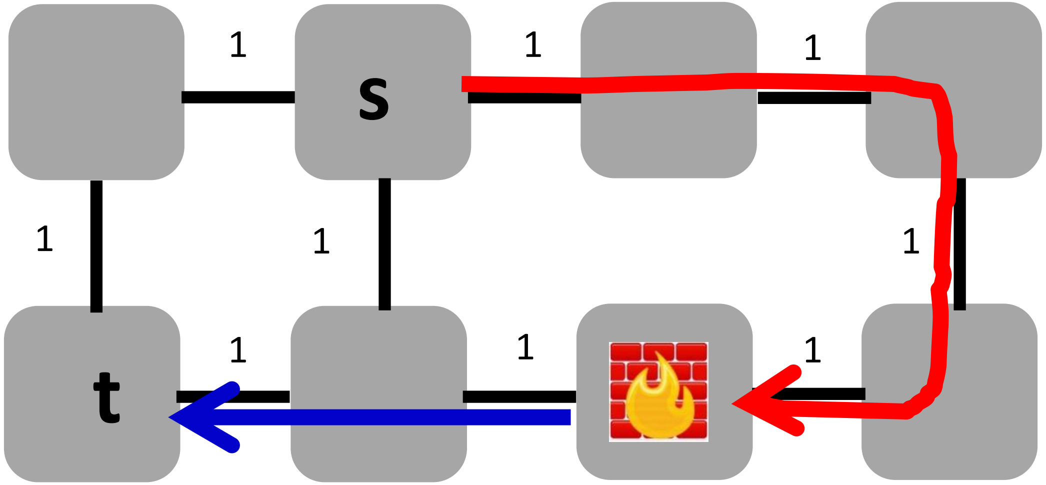
4+2=6

(Waypoint-)Routing is Hard

- Routing through a waypoint

- Greedy fails…

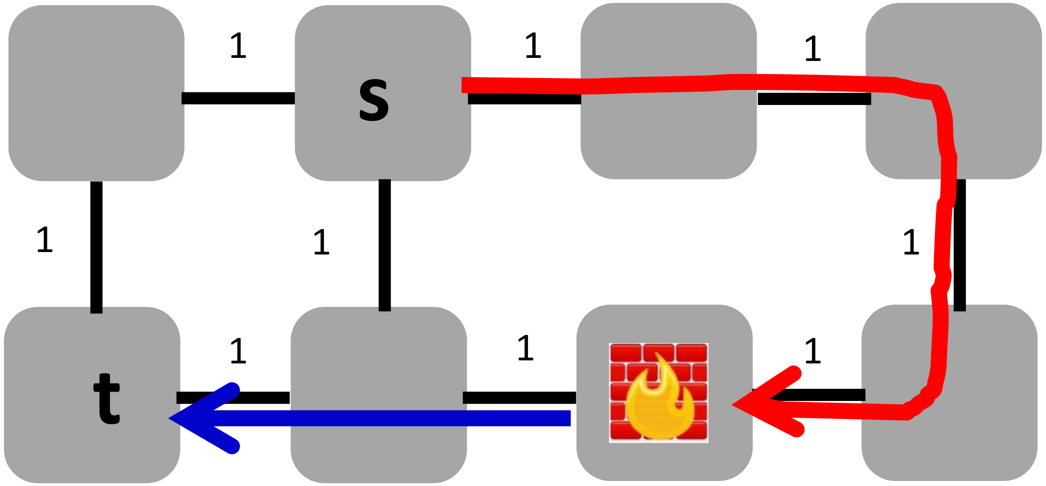
- ***NP-hard:*** reduction from edge-disjoint paths



Total length: 4+2=6

(Waypoint-)Routing is Hard

- Routing through a waypoint
- Greedy fails…
- ***NP-hard:*** reduction from edge-disjoint paths



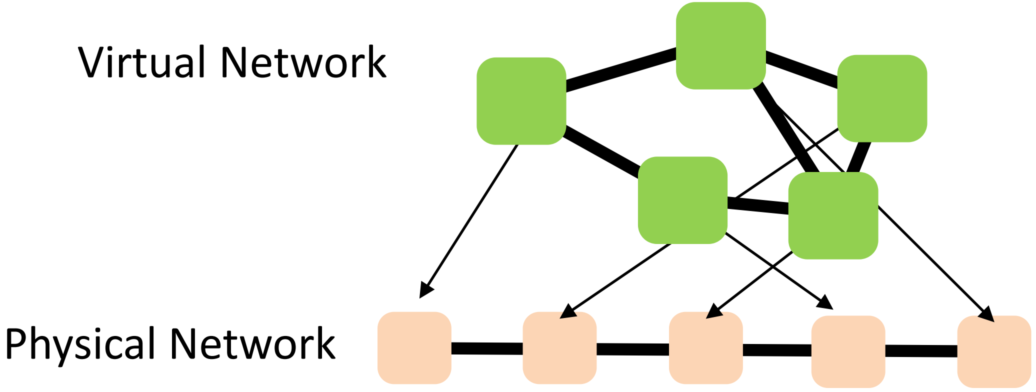
Total length:

4

Charting the Algorithmic Complexity of Waypoint Routing. ***SIGCOMM CCR*** 2018.

Embedding is Hard

- Embedding problems are often **NP-hard**



Generalization of ***Minimum Linear Arrangement*** (min sum embedding on a line)

Charting the Complexity Landscape of Virtual Network Embeddings ***IFIP Networking*** 2018.

DAN Design
~~Embedding~~ is Hard

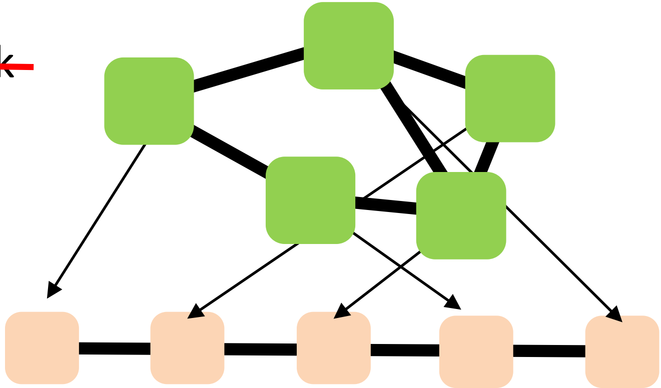
- Embedding problems are often **NP-hard**

Demand

~~Virtual Network~~

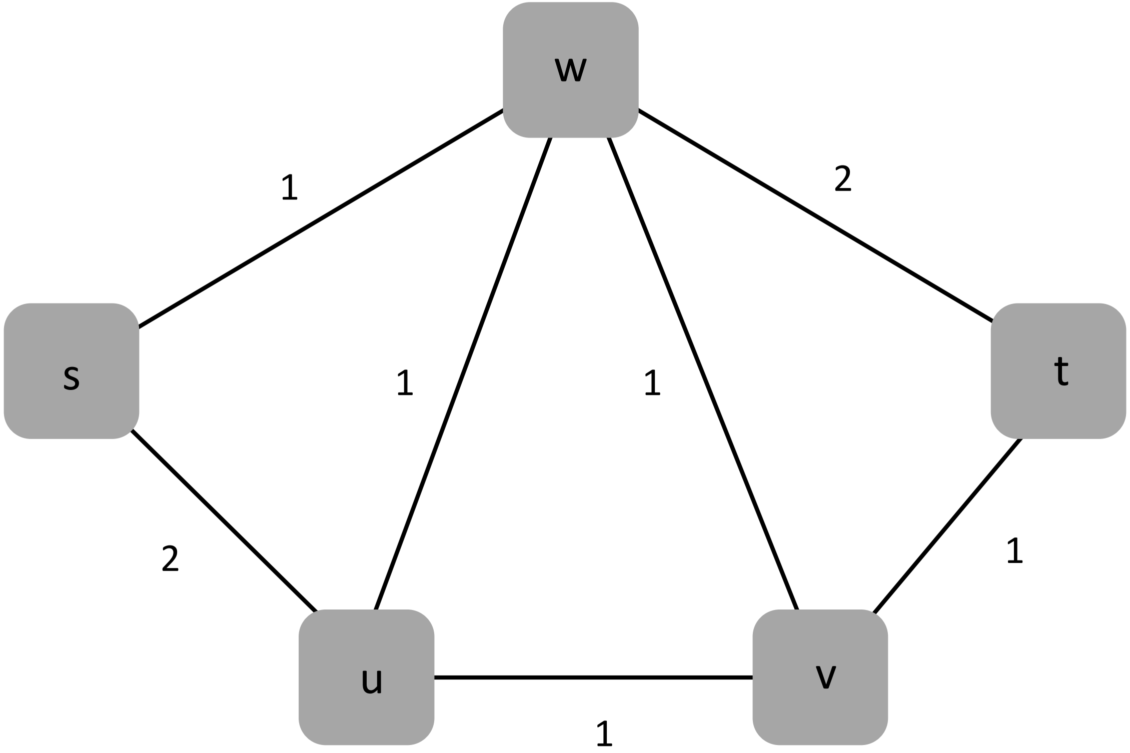
DAN (degree 2)

~~Physical Network~~

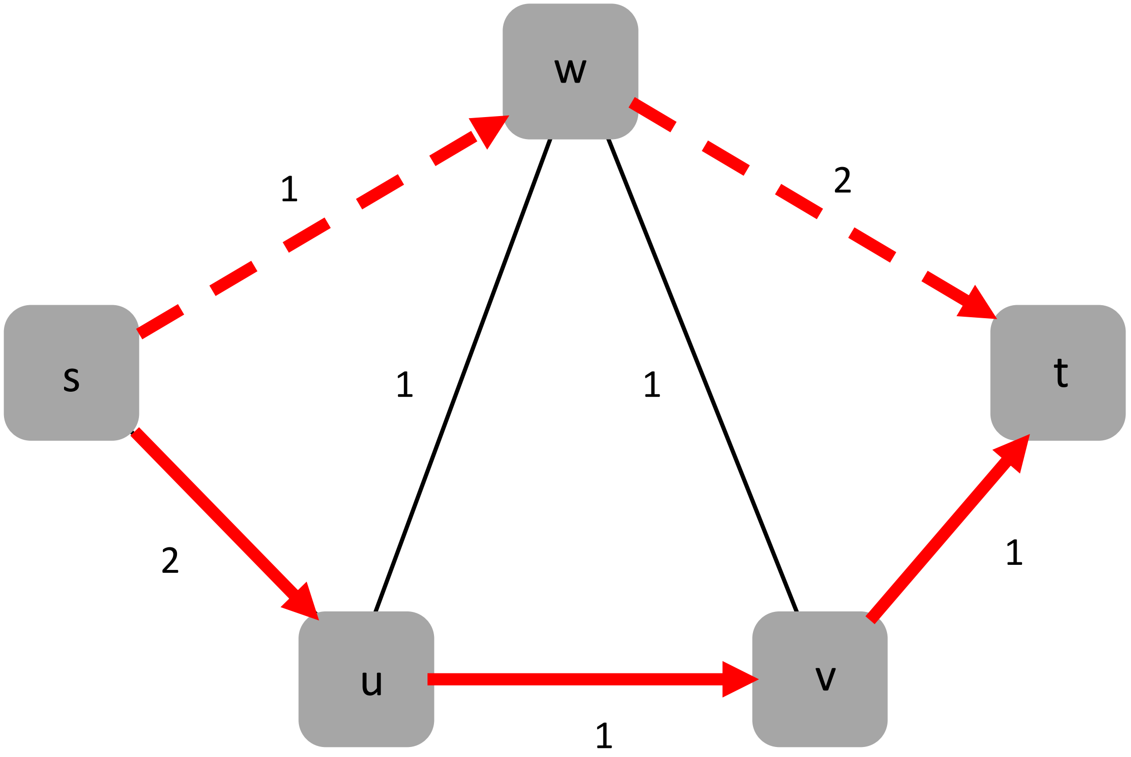


Generalization of ***Minimum Linear Arrangement*** (min sum embedding on a line)

Fast Flow Rerouting is Hard

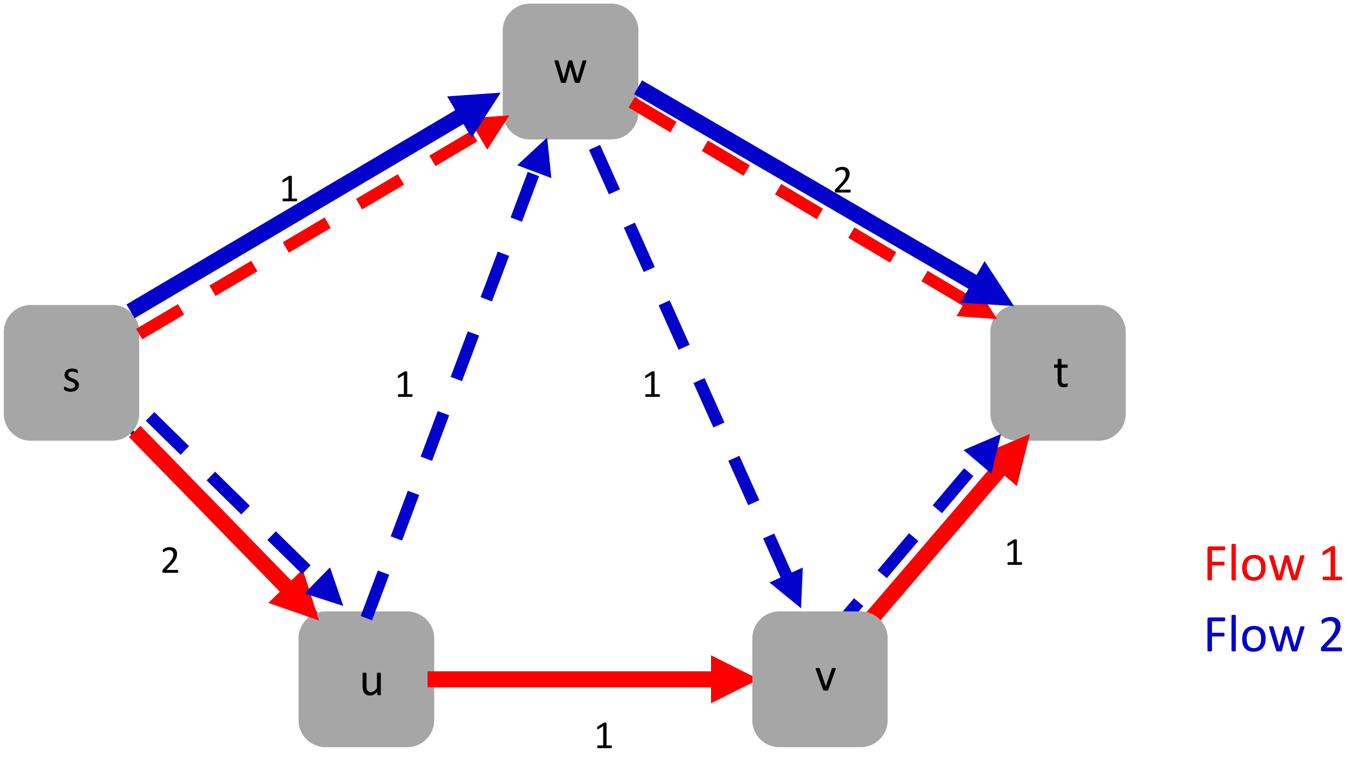


Fast Flow Rerouting is Hard

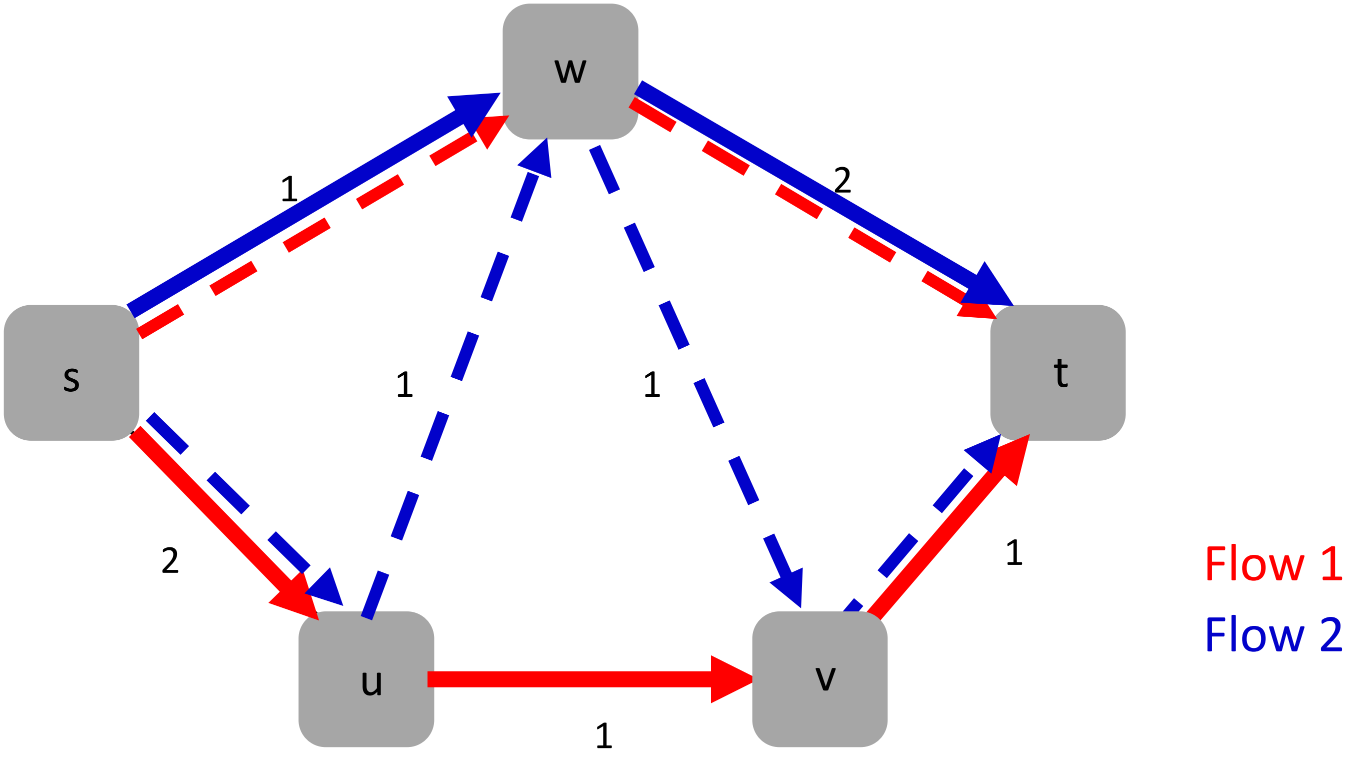


Flow 1

Fast Flow Rerouting is Hard



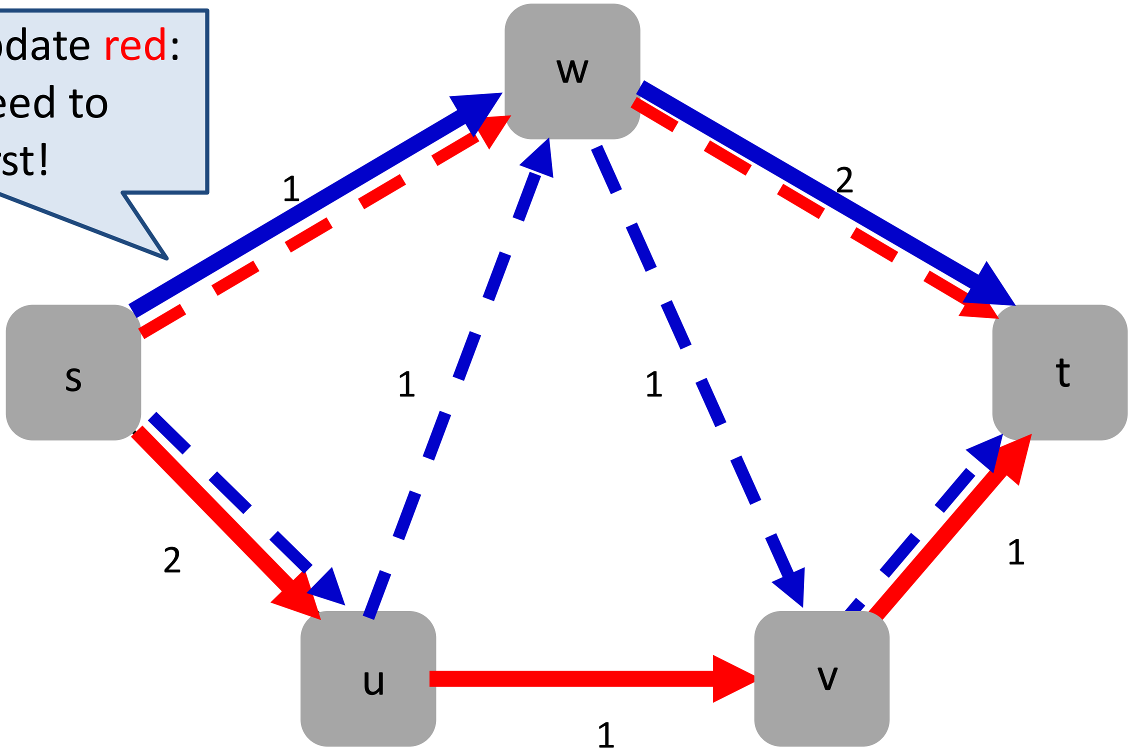
Fast Flow Rerouting is Hard



(Short) congestion-free update schedule?

Fast Flow Rerouting is Hard

e.g., cannot update red: congestion! Need to update blue first!



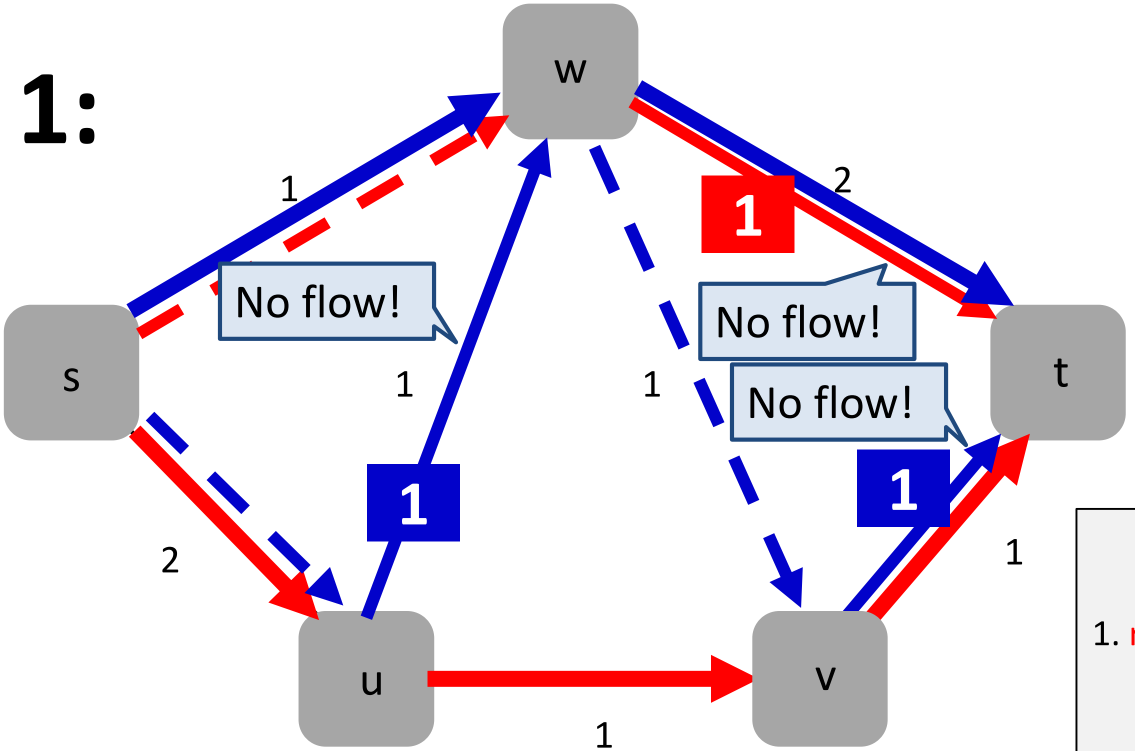
Flow 1

Flow 2

(Short) congestion-free update schedule?

Fast Flow Rerouting is Hard

Round 1:



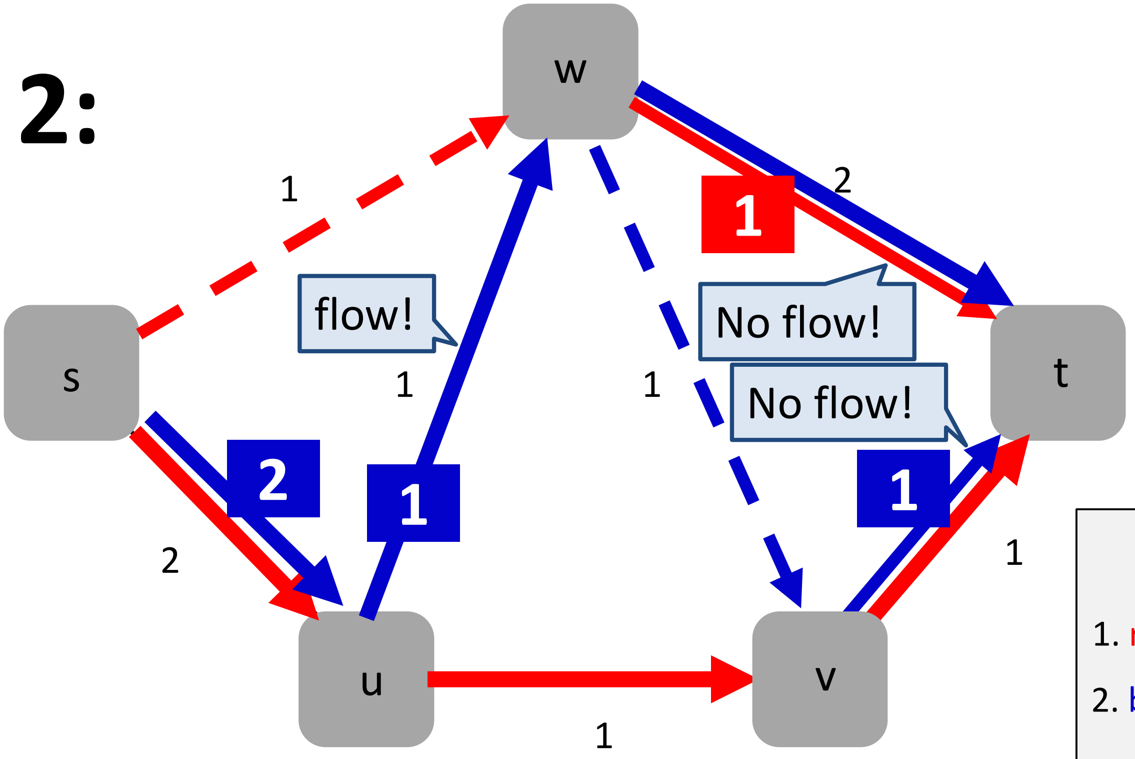
Prepare!

Schedule:

1. red@w,blue@u,blue@v

Fast Flow Rerouting is Hard

Round 2:

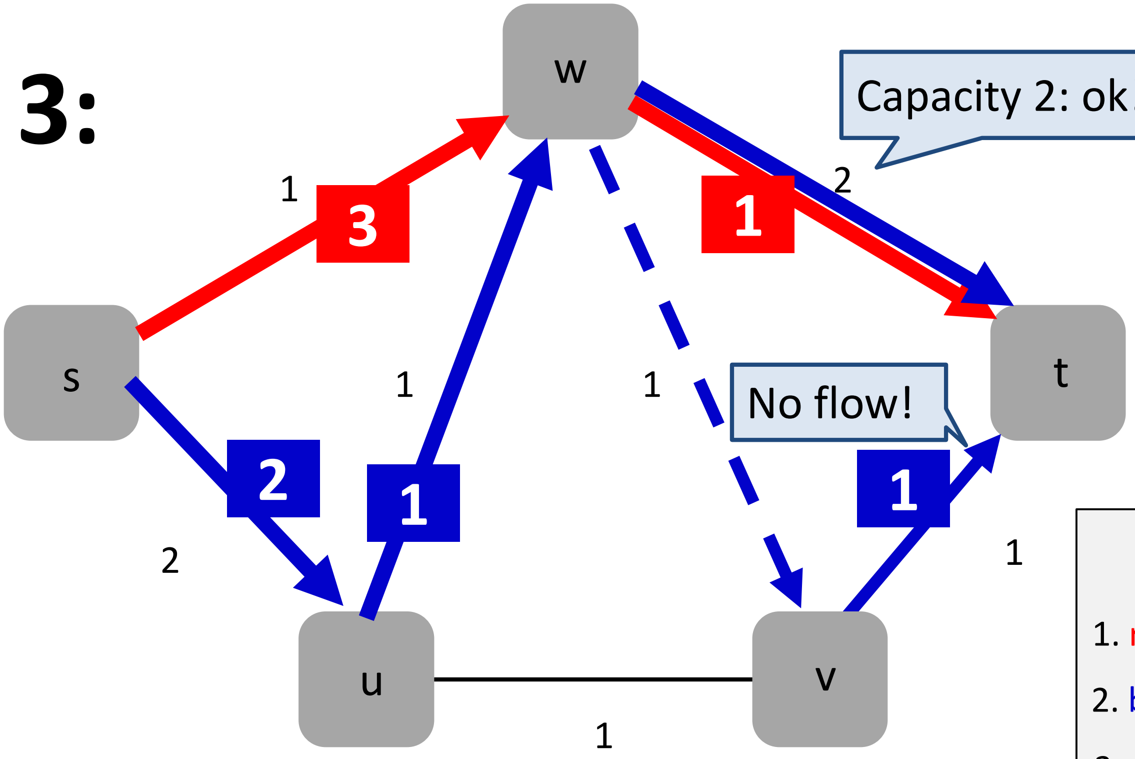


Schedule:

1. red@w,blue@u,blue@v
2. blue@s

Fast Flow Rerouting is Hard

Round 3:



Capacity 2: ok!

No flow!

Schedule:

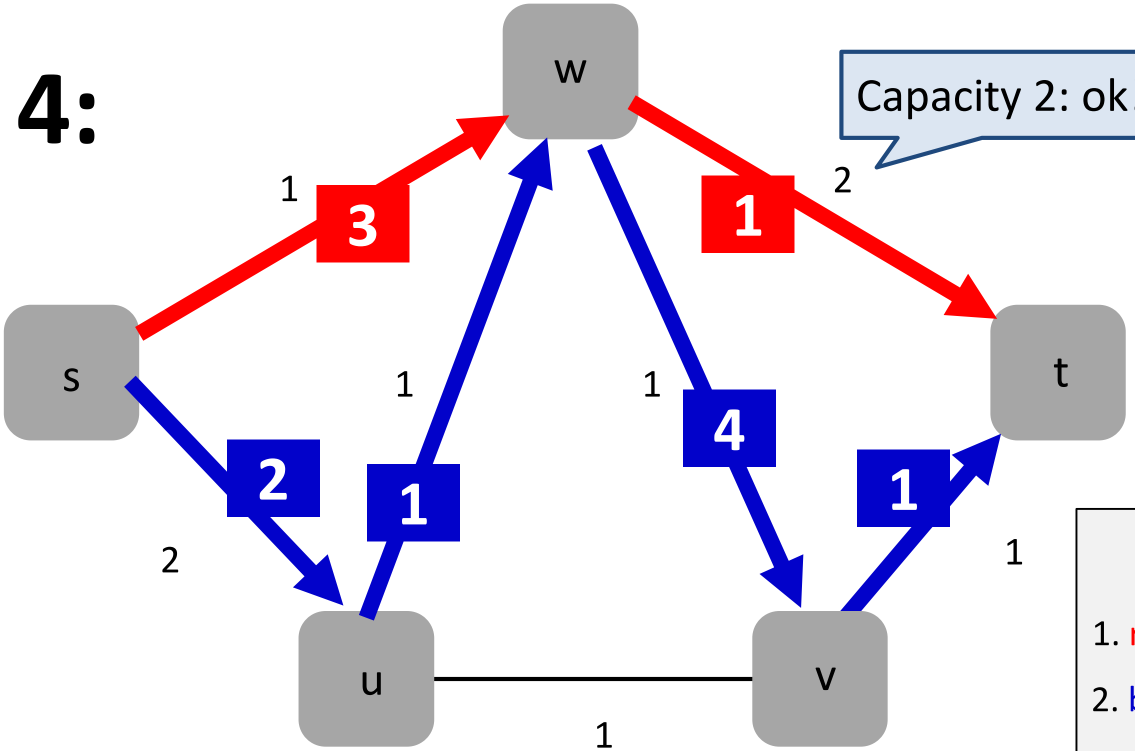
1. red@w,blue@u,blue@v

2. blue@s

3. red@s

Fast Flow Rerouting is Hard

Round 4:



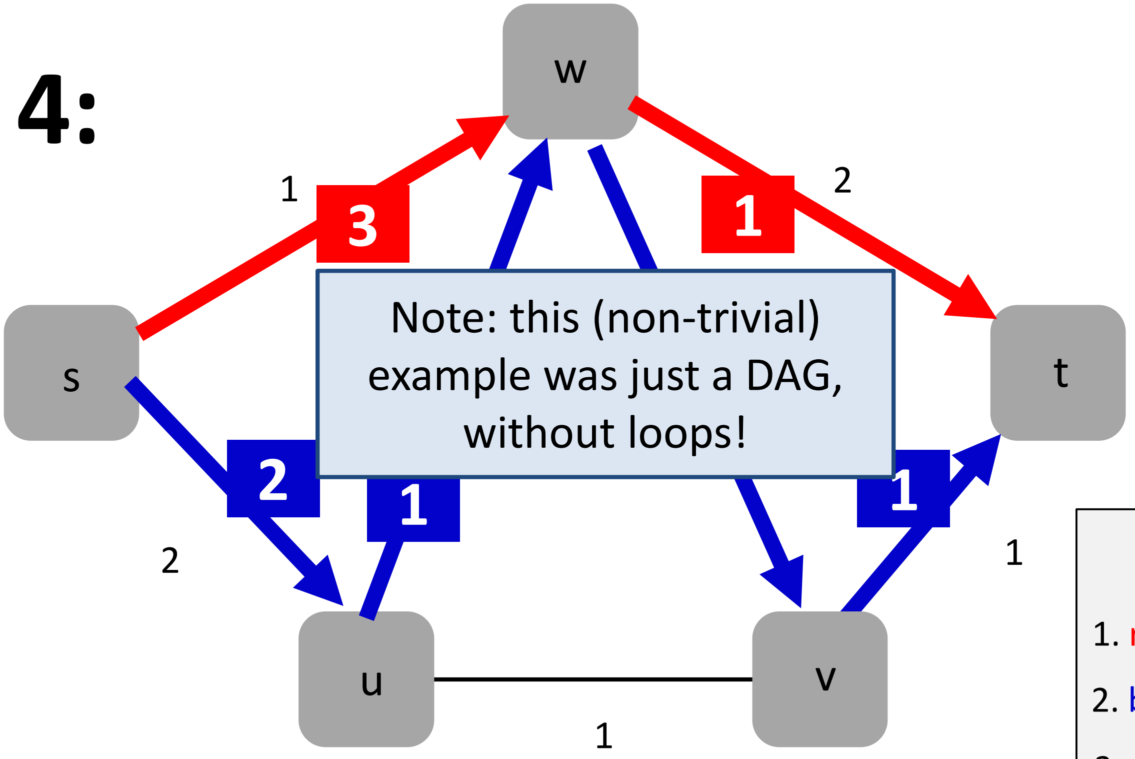
Capacity 2: ok!

Schedule:

1. red@w,blue@u,blue@v
2. blue@s
3. red@s
4. blue@w

Fast Flow Rerouting is Hard

Round 4:



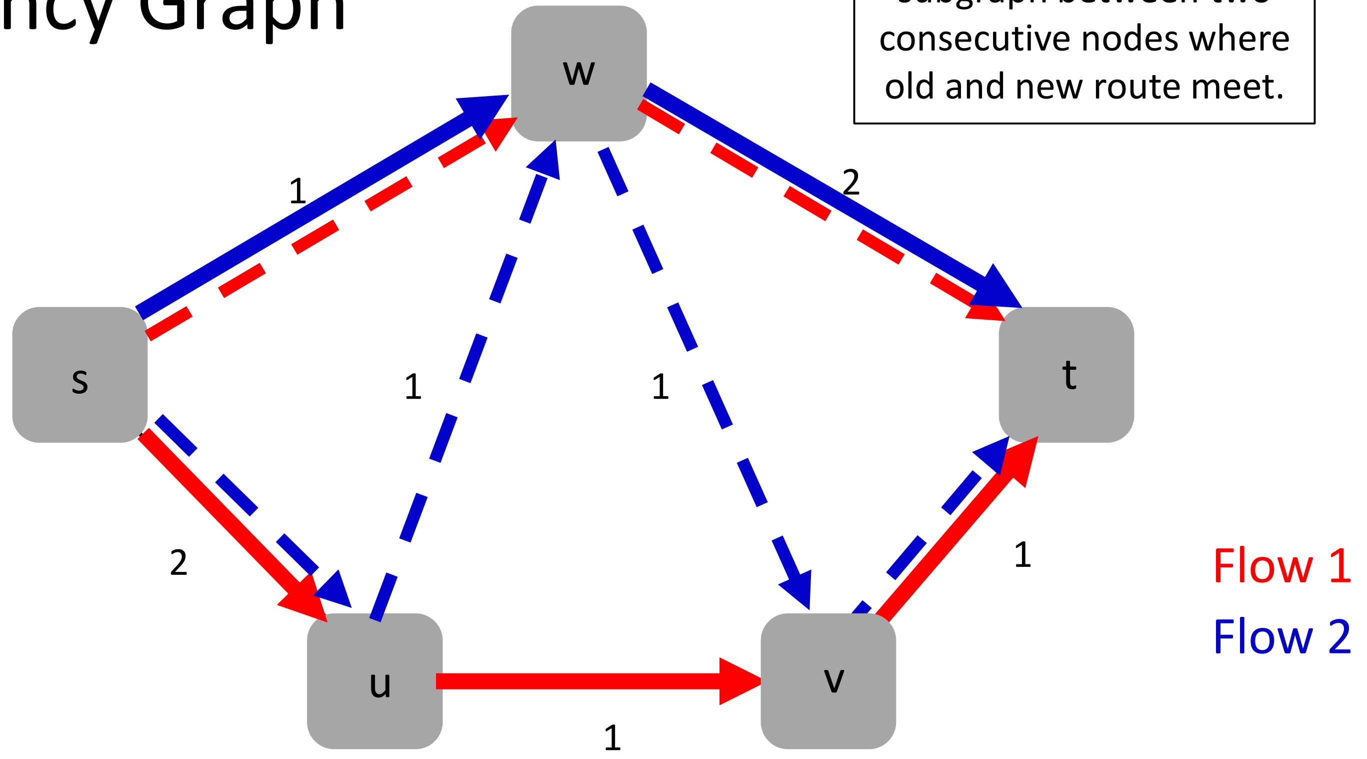
Note: this (non-trivial) example was just a DAG, without loops!

Schedule:

1. red@w,blue@u,blue@v
2. blue@s
3. red@s
4. blue@w

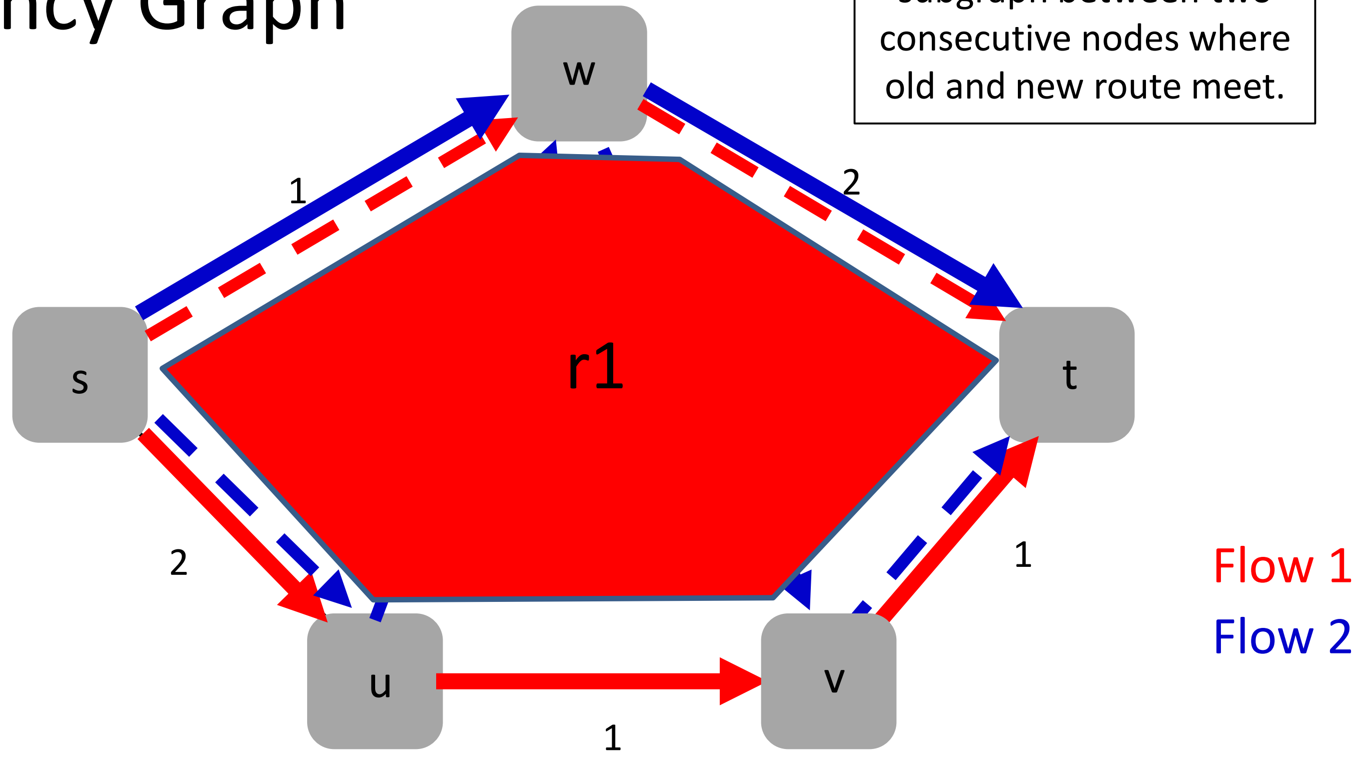
Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



Block Decomposition and Dependency Graph

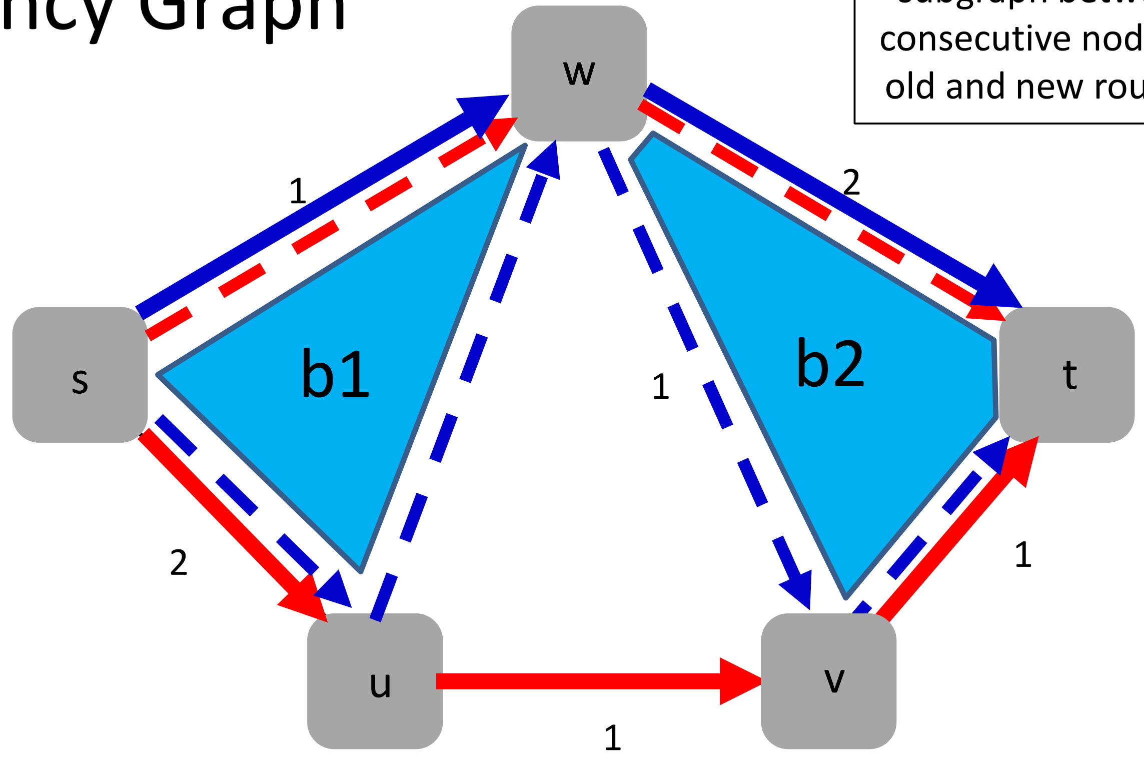
Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



Just one red block: r1

Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



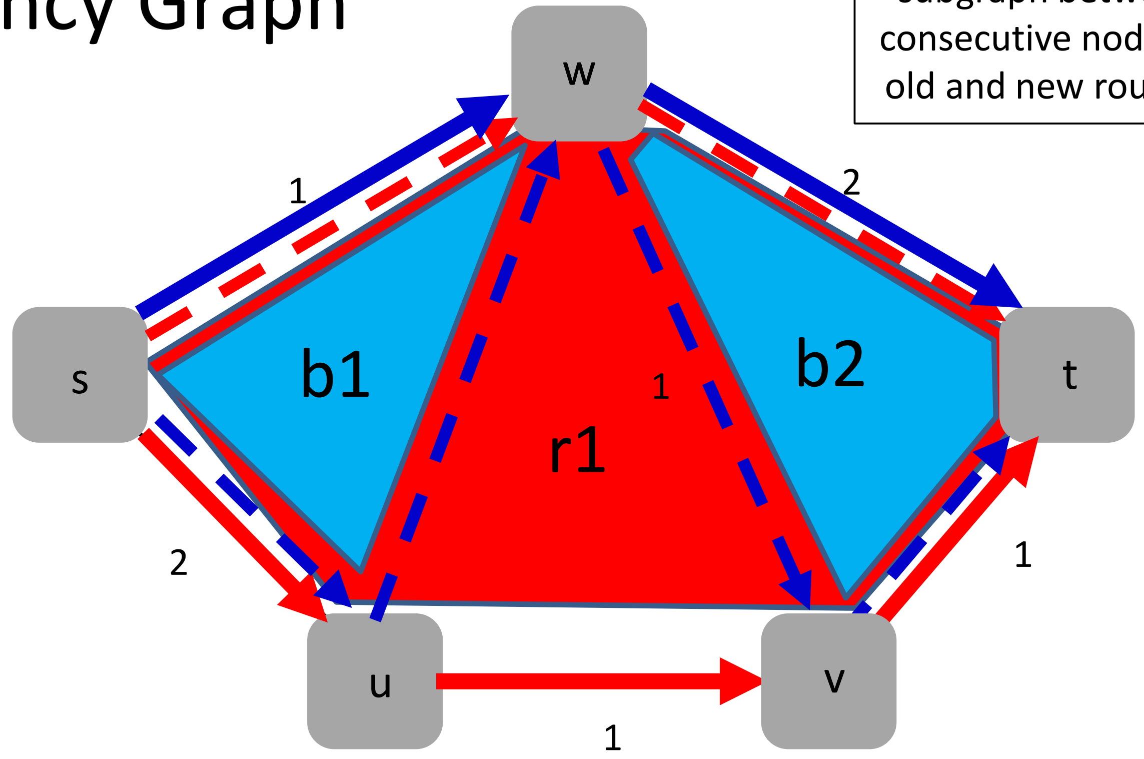
Flow 1

Flow 2

Two blue blocks: b1 and b2

Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



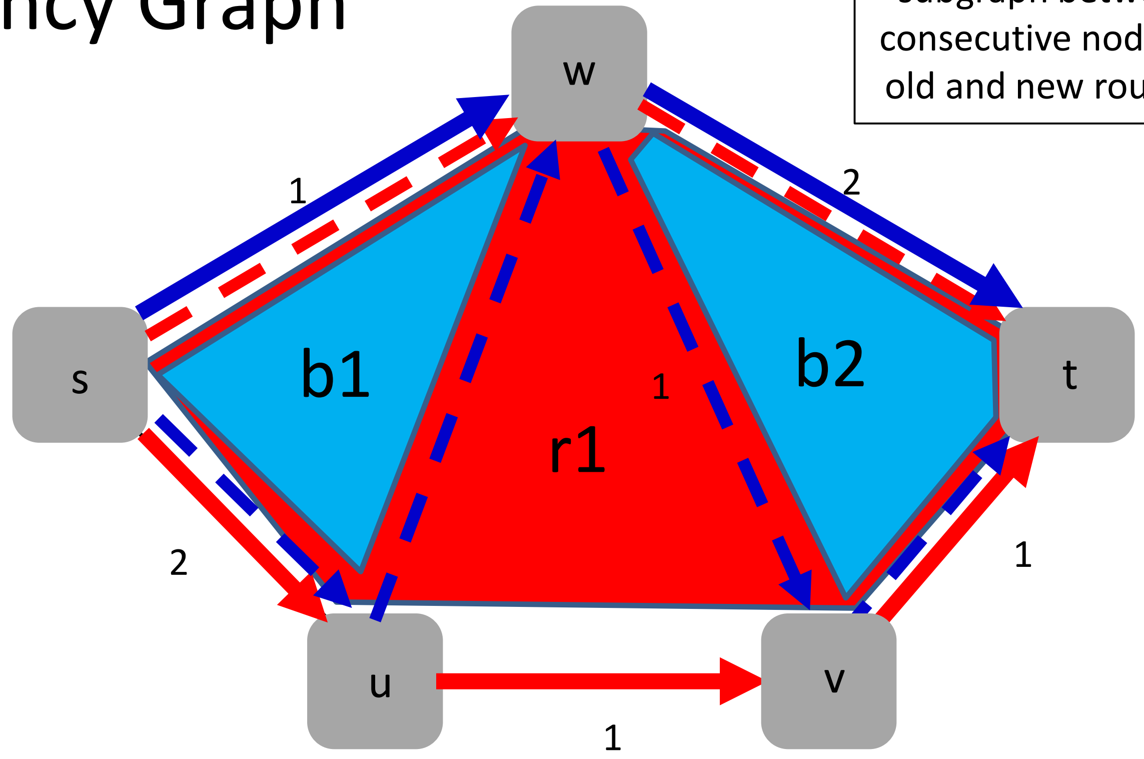
Flow 1

Flow 2

Dependencies: update b2 after r1 after b1.

Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



Flow 1

Flow 2

Dependencies: update b2 a

Congestion-Free Rerouting of Flows on DAGs. *ICALP* 2018.

Indeed: Exploiting Flexibilities is Even Hard for Computers

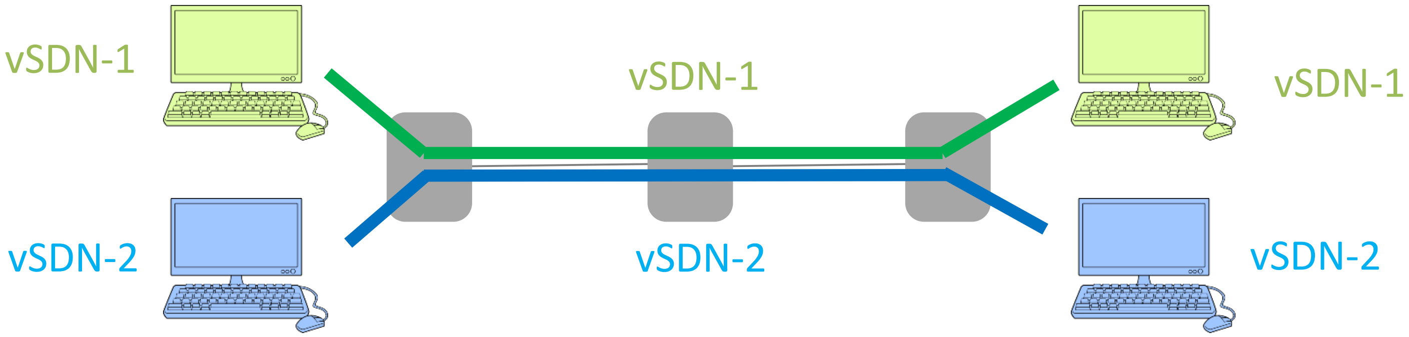
Part B: Because ***Reality***
(Modelling…) is Complex

Reality is Complex

Predictable performance is about more than just bandwidth reservation!

Reality is Complex

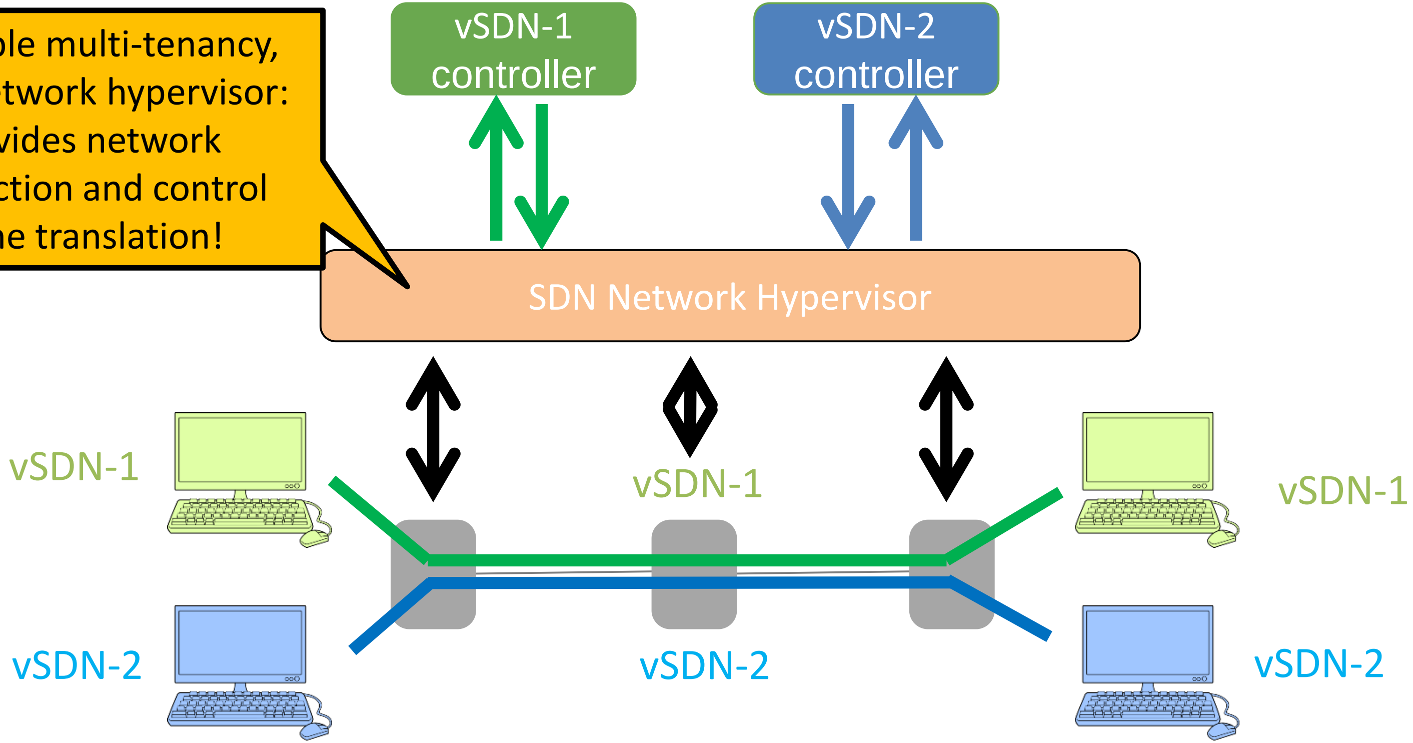
Predictable performance is about more than just bandwidth reservation!



An Experiment: 2 vSDNs with bw guarantee!

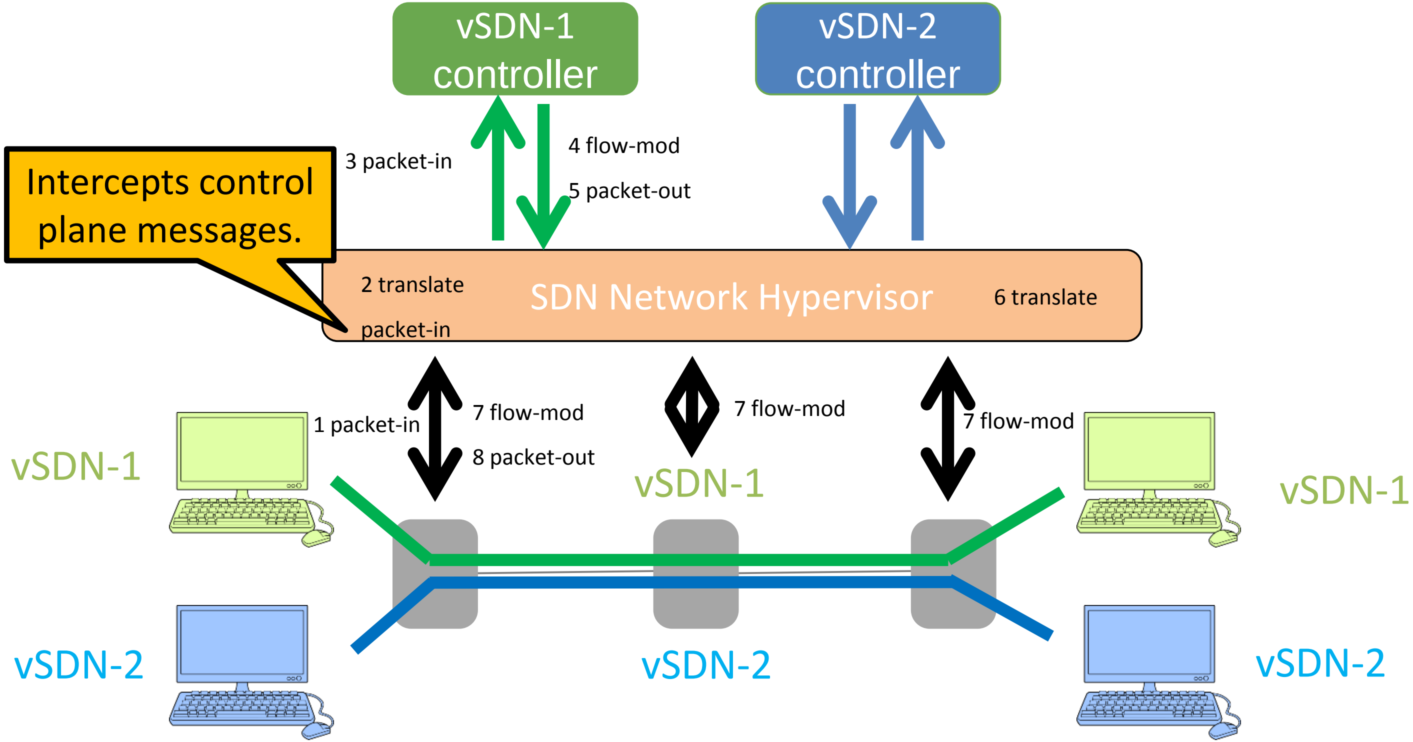
Reality is Complex

To enable multi-tenancy, need network hypervisor: provides network abstraction and control plane translation!



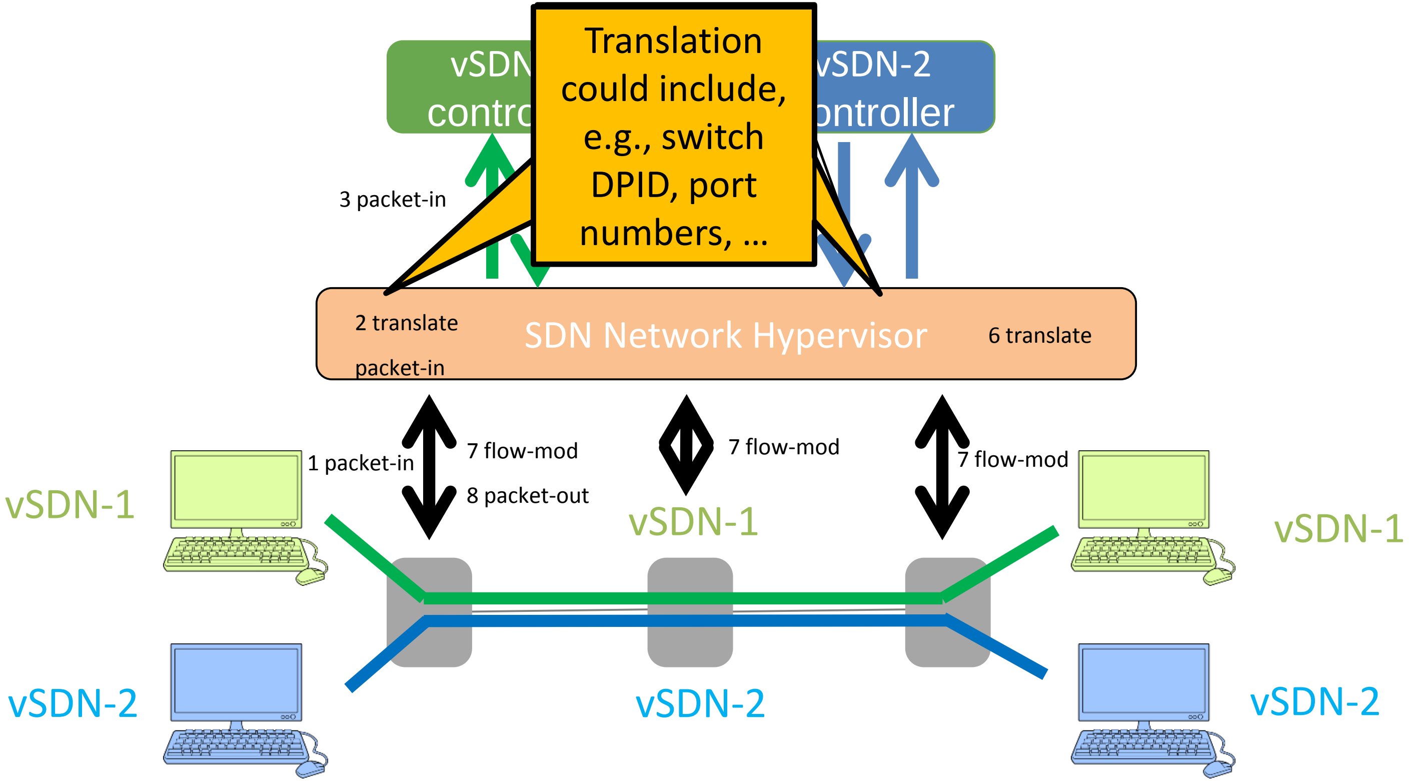
An Experiment: 2 vSDNs with bw guarantee!

Reality is Complex



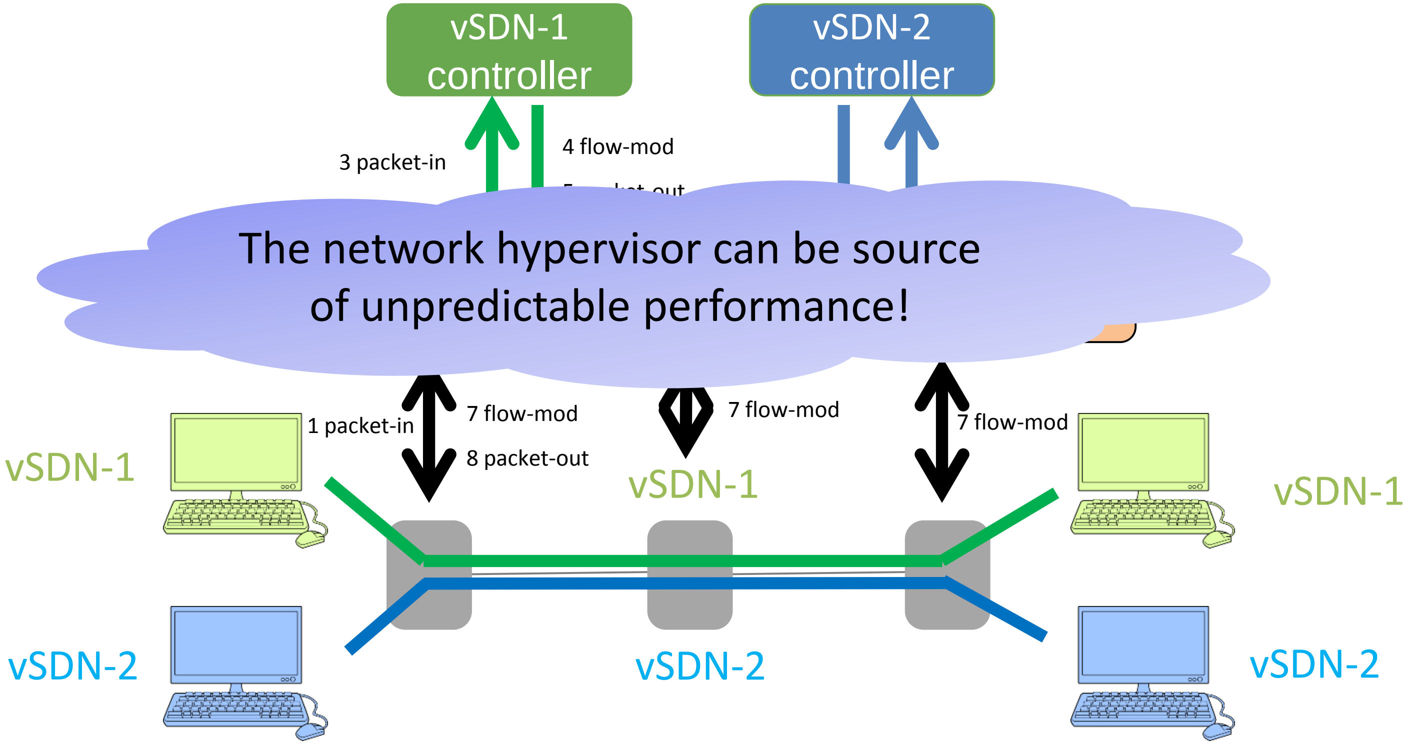
An Experiment: 2 vSDNs with bw guarantee!

Reality is Complex



An Experiment: 2 vSDNs with bw guarantee!

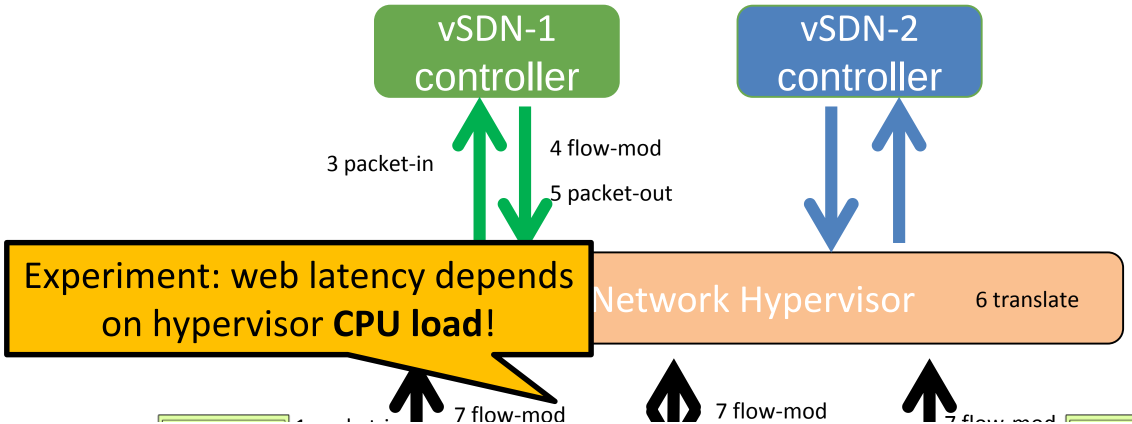
Reality is Complex



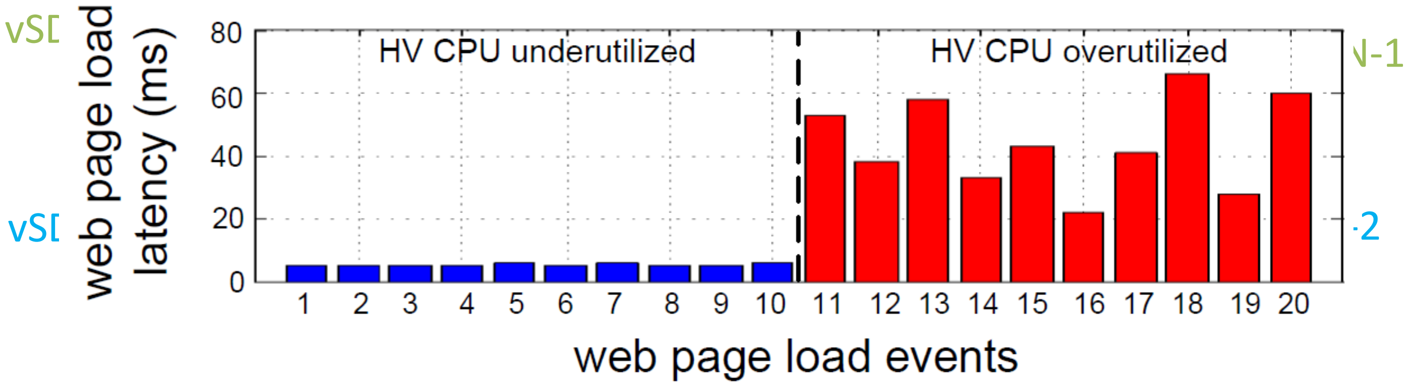
The network hypervisor can be source of unpredictable performance!

An Experiment: 2 vSDNs with bw guarantee!

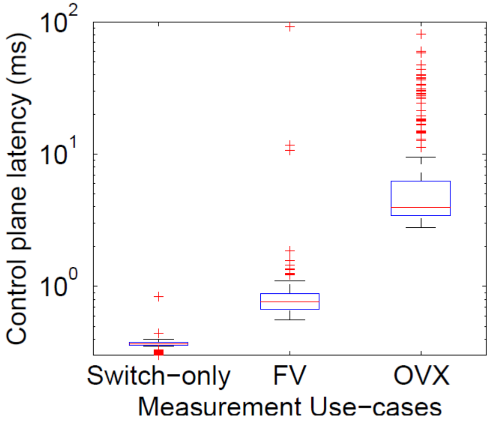
Reality is Complex



Experiment: web latency depends on hypervisor **CPU load**!



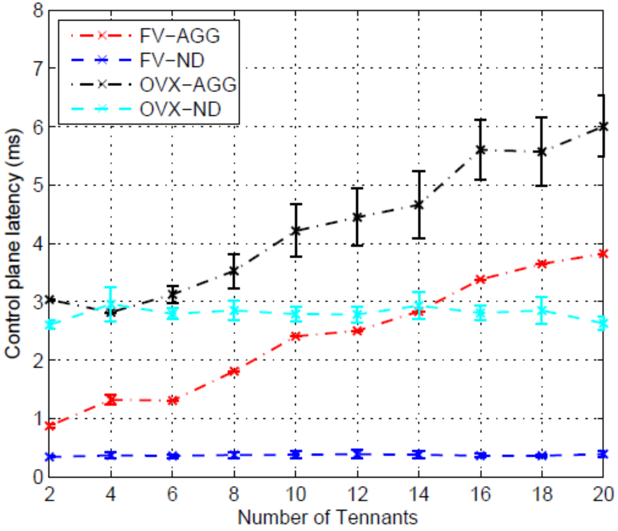
Need to Know Your Network Hypervisor



Performance also depends on hypervisor type…

(multithreaded or not, which version of Nagle's algorithm, etc.)

… number of tenants…



perfbench: A Tool for Predictability Analysis in Multi-Tenant Software-Defined Networks ***SIGCOMM Poster*** 2018.

Variance due to Algorithmic Complexity

- Seemingly similar **network configurations** can result in very different performance

- For example: match-action or **ACLs** rules which rely on **regular expressions**
  - Rule matching algorithm can have ***exponential runtime*** for some cases…
  - … while others are ***fast***
  - In addition: rules may ***overlap***

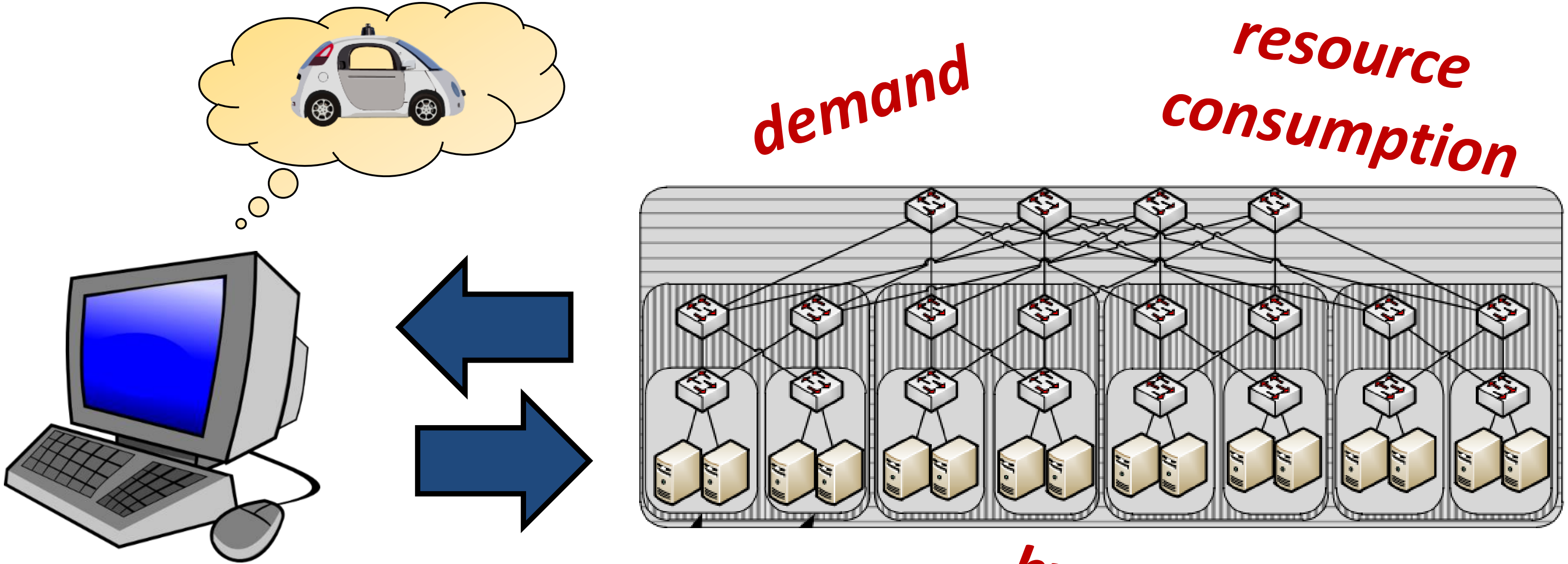
- OVS relies on **slow-/fast-path** mechanisms, depending on **flow caching** scheme performance can be very different

Policy Injection: A Cloud Dataplane DoS Attack.
SIGCOMM Demo 2018.

Indeed: Exploiting Flexibilities is Even Hard for Computers

The Case for
Demand/Interference/Resource/… ***-Aware***
aka. **Data-Driven** Networking and **ML**?!

“Demand/Interference/Resource/…” -Aware Networks



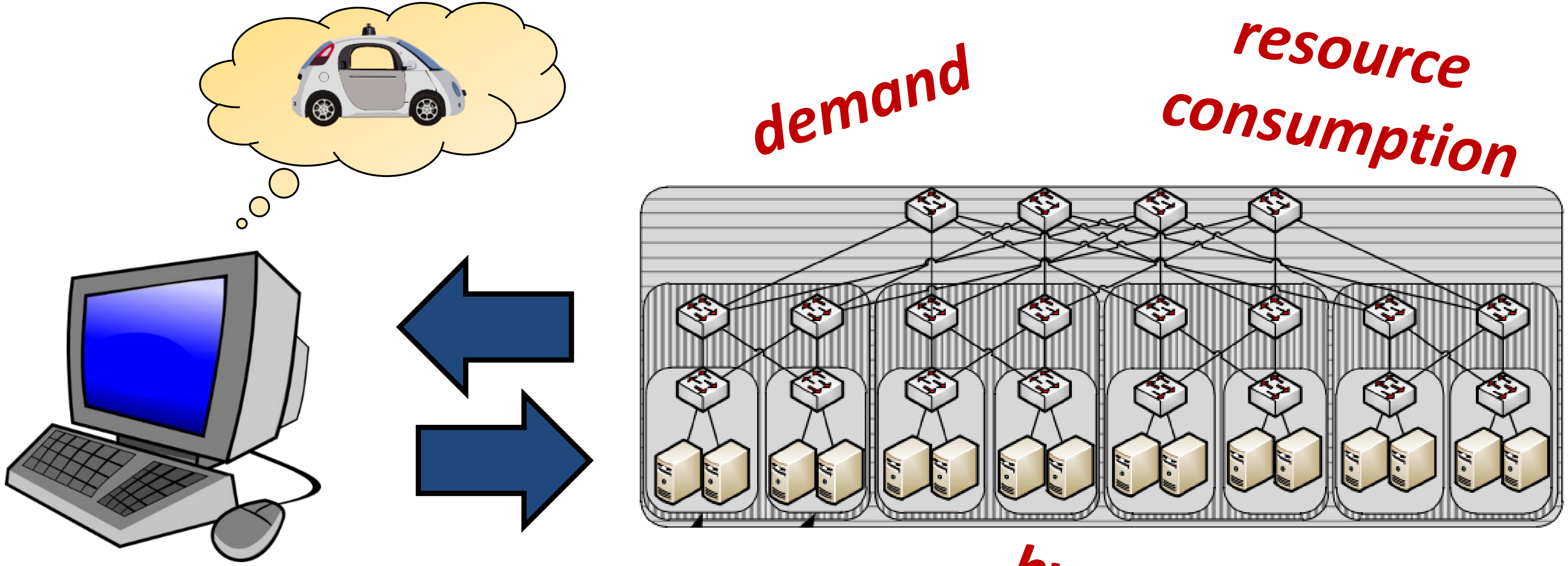
demand

resource consumption

measure
analyze
control

hypervisor interference

“Demand/Interference/Resource/…” -Aware Networks



demand

resource consumption

measure

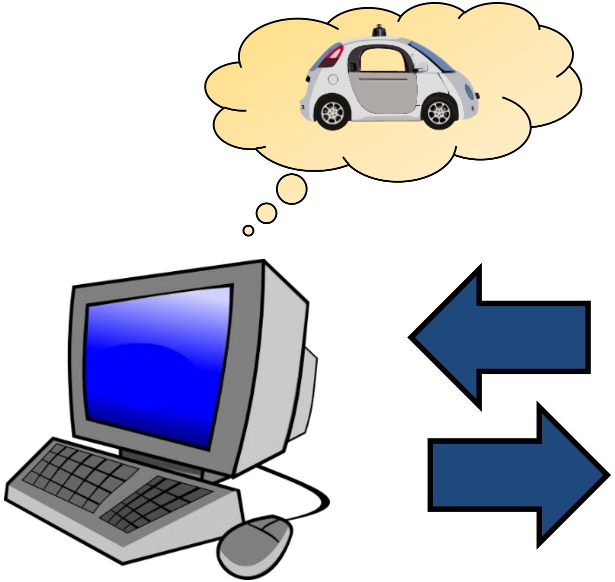
analyze

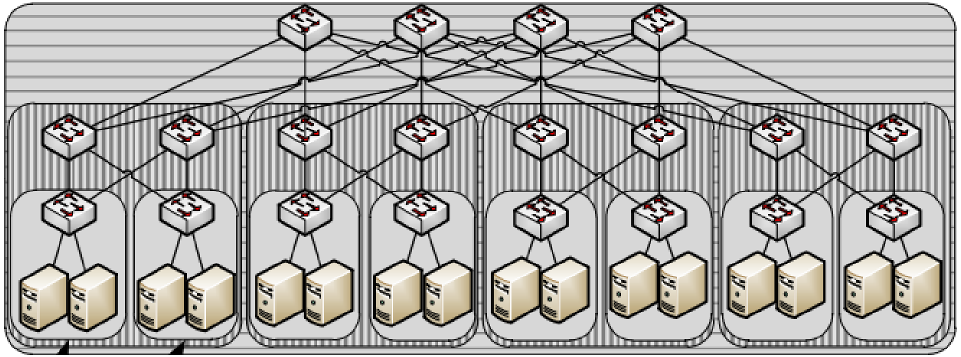
control

hypervisor interference

Allows to overcome worst-case lower bounds!

“Demand/Interference/Resource/…” -Aware Networks





demand

resource consumption

hyp

ference

measure
analyze
control

Allows to overcome worst-case lower boun

o'zapft is: Tap Your Network Algorithm's Big Data! ***Big-DAMA*** 2017.

NeuroViNE: A Neural Preprocessor for Your Virtual Network Embedding Algorithm. ***INFOCOM*** 2018.

What if there is no data?!

The Case for Empowerment

Empowerment

- **Empowerment**: infomation-theoretic measure how „***prepared***“ an agent is: can adapt to new environments
  - Known from ***robotics***

- Agent learns „ **different strategies**“, so becomes prepared

- If **objective function** or environment changes: change to different strategy

Empowering Self-Driving Networks.
SIGCOMM Wrksps 2018.

Roadmap

Nils Bohr

Thank you! ☺

- Predictable performance under **uncertainty** is hard

- Observation: at the same time, networks become more **flexible**! Idea: exploit for ***predictability***…

Flexibility

- … but it can be ***hard for humans***:

a case for **formal methods**? *Hot* right now (and here!)

- … but that can even be ***hard for computers***: so?!

Especially **quantitative aspects** but important for QoE!

Further Reading

Demand-Aware Network Designs of Bounded Degree
Chen Avin, Kaushik Mondal, and Stefan Schmid.
31st International Symposium on Distributed Computing (**DISC**), Vienna, Austria, October 2017.

Characterizing the Algorithmic Complexity of Reconfigurable Data Center Architectures
Klaus-Tycho Foerster, Monia Ghobadi, and Stefan Schmid.
ACM/IEEE Symposium on Architectures for Networking and Communications Systems (**ANCS**), Ithaca, New York, USA, July 2018.

SplayNet: Towards Locally Self-Adjusting Networks
Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker.
IEEE/ACM Transactions on Networking (**TON**), Volume 24, Issue 3, 2016.

Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks
Stefan Schmid and Jiri Srba.
37th IEEE Conference on Computer Communications (**INFOCOM**), Honolulu, Hawaii, USA, April 2018.

WNetKAT: A Weighted SDN Programming and Verification Language
Kim G. Larsen, Stefan Schmid, and Bingtian Xue.
20th International Conference on Principles of Distributed Systems (**OPODIS**), Madrid, Spain, December 2016.

Charting the Complexity Landscape of Virtual Network Embeddings
Matthias Rost and Stefan Schmid. **IFIP Networking**, Zurich, Switzerland, May 2018.

Virtual Network Embedding Approximations: Leveraging Randomized Rounding
Matthias Rost and Stefan Schmid. **IFIP Networking**, Zurich, Switzerland, May 2018.

Logically Isolated, Actually Unpredictable? Measuring Hypervisor Performance in Multi-Tenant SDNs
Arsany Basta, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. ArXiv Technical Report, May 2017.

Empowering Self-Driving Networks
Patrick Kalmbach, Johannes Zerwas, Peter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid.
ACM SIGCOMM 2018 Workshop on Self-Driving Networks (**SDN**), Budapest, Hungary, August 2018.

See also references on slides!